Uniwersytet Łódzki

Wydział Matematyki i Informatyki

Michał Bieńkowski

Numer albumu 289475

Wysokopoziomowy silnik gier dla urządzeń mobilnych z systemem Android

Praca magisterska
napisana pod kierunkiem:
dr Piotra Fulmańskiego
Katedra Analizy Matematycznej
i Teorii Sterowania

Łódź 2013

Spis treści

1 Wstęp **1**

1.1 Cel oraz zakres pracy 1

1.2 Informacje podstawowe 2

1.2.1 Stosowane konwencje 2

1.2.2 Ważne pojęcia 3

1.2.2.1 Aplikacja czasu rzeczywistego 4

1.2.2.2 Aplikacja multimedialna 4

1.2.2.3 Gra komputerowa 4

1.2.2.4 Silnik gier komputerowych 4

2 Biblioteka OpenGL **5**

2.1 Historia powstania 5

2.2 Sposób działania 6

3 Platforma Android **11**

3.1 Historia powstania 11

3.2 Dlaczego Android? 12

3.3 Programowanie aplikacji mobilnej 12

3.3.1 Podstawy 12

3.3.2 Obsługa biblioteki OpenGL 14

3.3.3 Wyświetlanie podstawowych obiektów 17

3.3.4 Teksturowanie obiektów 18

4 Silnik gier **24**

4.1 Klasy oraz przykłady użycia . . . 25

4.1.1 CatEngineCore . . . 26

4.1.2 Typy prymitywne . . . 28

4.1.2.1 Color . . . 29

4.1.2.2 Point . . . 29

4.1.2.3 Size . . . 30

4.1.2.4 Vector . . . 31

4.1.3 Typy renderowane . . . 31

4.1.3.1 Renderable . . . 32

4.1.3.2 Kształty . . . 33

4.1.3.2.1 Shape . . . 33

4.1.3.2.2 Line . . . 34

4.1.3.2.3 Rectangle . . . 34

4.1.4 Elementy graficzne . . . 35

4.1.4.1 Sprite . . . 35

4.1.4.2 AnimatedSprite . . . 36

4.1.5 Elementy tekstowe . . . 39

4.1.5.1 Text . . . 39

4.1.6 Scena oraz jej elementy . . . 41

4.1.6.1 Scene . . . 42

4.1.6.2 Layer, LayerTree . . . 47

4.1.6.3 Camera . . . 49

4.1.6.4 TouchPointer . . . 50

4.1.7 Obsługa tekstur . . . 50

4.1.7.1 Texture . . . 51

4.1.7.2 TextureRegion . . . 53

4.1.7.3 TextureAnimation 54
4.1.7.4 Font 55
4.1.8 Obsługa dźwięków 58
4.1.8.1 Audio 59
4.1.9 Klasy ogólnodostępne 60
4.1.9.1 Handlery 60
4.1.9.1.1 TouchHandler 60
4.1.9.2 Przydatne narzędzia 62
4.1.9.2.1 Interpolator 62
4.1.9.2.2 FPSCounter 63
4.1.9.3 Interfejsy 64
4.1.9.3.1 IColorable 64
4.1.9.3.2 IDisposable 64
4.1.9.3.3 IManager 64
4.1.9.3.4 IMovable 65
4.1.9.3.5 ISizable 65
4.1.9.3.6 IRotatable 66
4.1.9.3.7 IRenderable 66
4.1.9.3.8 IShape 67
4.1.9.3.9 ISprite 67
4.1.9.4 Listenery 68
4.1.9.4.1 AnimationListener 68
4.1.9.4.2 AudioListener 68
4.1.9.4.3 CameraListener 68
4.1.9.4.4 ColorListener 69
4.1.9.4.5 PositionListener 69
4.1.9.4.6 RotationListener 69

4.1.9.4.7 ScaleListener 69

4.1.9.4.8 InterpolatorListener 69

4.1.9.5 Managery 70

5 Podsumowanie **71**

6 Załączniki do pracy **72**

6.1 Spis tabel 72

6.2 Spis rysunków 72

6.3 Spis kodów źrodłowych 73

6.4 Słownik pojęć 75

7 Bibliografia **76**

Rozdział 1

Wstęp

Tematyka gier komputerowych towarzyszy informatce od długiego czasu. Lata 1947 - 1972 określa się jako rozwój gier pierwszej generacji. W roku 1947 powstał analogowy symulator pocisku rakietowego używający lamp elektronowych do obliczenia trajektorii lotu. Był on uważany za prototyp pierwszego programu rozrywkowego. Z kolei za początek graficznych aplikacji uważa się rok 1952, w którym stworzona została gra *Noughts and Crosses*. Gra ta jest powszechnie znana w Polsce jako *kółko i krzyżyk*, a jej pierwsza graficzna wersja została stworzona w ramach pracy doktorskiej, której autorem był Alexander Sandy Douglas. Natomiast pierwszą grą, która odniosła sukces komercyjny był *Pong* z roku 1972, wydawany przez firmę *Atari*.

Obecnie gry znacznie różnią się od tych znanych ponad 40 lat temu. Na przestrzeni tych kilkudziesięciu lat technologia poszła do przodu, a wraz z nią popularność urządzeń takich jak komputer, tablet lub smartfon. Co za tym idzie zwiększyło się zapotrzebowanie na wszelakiego rodzaju gry na PC lub urządzenia mobilne. Obecnie stare gry takie jak *Pong* lub *Noughts and Crosses* przeżywają swoją drugą młodość. Każda osoba bez względu na wiek lubi pograć, chociażby aby się odstresować i zapomnieć o otaczającej rzeczywistości. Dlatego gry zawsze będą cieszyły się ogromną popularnością.

1.1 Cel oraz zakres pracy

Celem mojej pracy było napisanie silnika gier dla urządzeń mobilnych posiadających system operacyjny Android.

Pomysł stworzenia silnika gier powstał w momencie, gdy chciałem napisać prostą grę.

Chciałem wówczas spróbować swoich sił wykorzystując silnik, który z mojej strony nie wymagałby jakichkolwiek nakładów pieniężnych. Znalazłem kilka silników, których forma była na tyle skomplikowana, iż nie pozwalałaby na stworzenie jakiejkolwiek gry w sensownych ramach czasowych. Pozostałe silniki, które w miarę odpowiadały moim oczekiwaniom posiadały czasochłonną konfigurację oraz brak dokumentacji, co było szczególnym utrudnieniem w tworzeniu aplikacji. Doszedłem do wniosku, że najlepiej napisać własny silnik gier, który będzie spełniał moje wymagania i może ułatwi pracę komuś innemu. Co za tym idzie powinien być szybki, intuicyjny oraz wygodny w użytkowaniu.

W związku ze zwiększającą się popularnością wszelkiego rodzaju urządzeń przenośnych pracujących na systemie Android postanowiłem stworzyć silnik gier na tą platformę. W ciągu zaledwie kilku lat powstało bardzo dużo gier na telefony komórkowe lub tablety. Część z nich nie jest zbyt skomplikowana, ale przyciąga bardzo dużą ilość graczy. Co za tym idzie mój silnik mógłby w znaczny sposób pomóc w napisaniu prostej aplikacji.

W swojej pracy przedstawiam gotowy silnik wraz z pełną dokumentacją oraz kilkoma przykładami w jakich miałby zastosowanie.

1.2 Informacje podstawowe

Zakładam, iż czytelnik niniejszej pracy posiada podstawową wiedzę z zakresu programowania w języku Java oraz środowisk Android i OpenGL. Jest to bowiem niezbędne do pełnego zrozumienia opisów oraz fragmentów kodu przedstawionych w pracy.

1.2.1 Stosowane konwencje

W niniejszej pracy zostały zastosowane następujące konwencje typograficzne:

- *Kursywa*:
  - nowe pojęcia,
  - słowa kluczowe,
  - adresy URL,
  - nazwy plików,
  - rozszerzenia plików,

  - ścieżki do plików.

- `Czcionka o stałej szerokości`:
  - kod źródłowy,
  - odwołania do metod oraz zmiennych.

Konwencje dotyczące kodu źródłowego są następujące:

- Nazewnictwo funkcji i zmiennych:
  - dla większości nazwa stosowany jest standard camelCase,

```
1 public void onPointerDown(TouchPointer pointer);
```

  - dla zmiennych o stałych wartościach używane są wielkie litery,

```
1 final int STALA_WARTOSC = 100;
```

  - prywatne oraz chronione zmienne posiadają przedrostek m,

```
1 private Camera mCamera;
2 protected String mSceneName;
```

  - każde związane ze sobą instrukcje kodu są oddzielone od innych jedną pustą linią.

```
1  float x = vector.x;
2  float y = vector.y;
3
4  for (int i = 0; i < vertices.length; i += 2)
5  {
6      vertices[i] += x;
7      vertices[i + 1] += y;
8  }
9
10 updateBuffer();
```

1.2.2 Ważne pojęcia

Tworzenie silnika gier wiąże się ze stworzeniem bibliotek oraz gotowych narzędzi programistycznych, które mają za zadanie ułatwić napisanie multimedialnej aplikacji czasu rzeczywistego wykorzystującej wsparcie graficzne. Na wstępie chciałbym krótko przedstawić i wyjaśnić czym są wyżej wymienione pojęcia.

1.2.2.1 Aplikacja czasu rzeczywistego

Aplikacja czasu rzeczywistego *(eng. real-time application)* jest specyficznym programem, który swe działania wykonuje w określonych ramach czasowych. Zazwyczaj wykorzystuje ona nieskończoną pętlę, w której aktualizuje dane oraz przy okazji przechwytując akcje użytkownika (o ile takie wystąpiły) w zadanym interwale czasowym. Przeciwieństwem wyżej opisanej aplikacji jest aplikacja oparta na wykonywaniu określonych operacji dopiero w momencie wykrycia akcji użytkownika *(eng. event-based application)*.

1.2.2.2 Aplikacja multimedialna

Aplikacje multimedialne wykorzystują między innymi takie elementy jak tekst, grafika, animacje lub dźwięk. Bardzo często wykorzystywane są również interakcje z użytkownikiem w celu wygenerowania odpowiednich efektów na ekranie. Tego typu aplikacje w dużo lepszy sposób pomagają osobom skupić się na odbieranych informacjach oraz wczuć się w dane środowisko.

1.2.2.3 Gra komputerowa

Istnieje wiele definicji czym jest gra komputerowa. Według mnie gra jest to pewnego rodzaju aplikacja multimedialna wykonywana w czasie rzeczywistym, która cechuje się dużą interaktywnością z użytkownikiem, głównie w celach rozrywkowych. Nie wszystkie gry tworzy się tylko i wyłącznie dla rozrywki, część z nich pozwala nauczyć się nowych rzeczy takich jak języki obce (gry edukacyjne).

1.2.2.4 Silnik gier komputerowych

Silnik gier komputerowych jest to zbiór narzędzi programistycznych, dzięki któremu można w łatwy sposób zarządzać zasobami urządzenia oraz elementami sceny. Ułatwiają to gotowe metody, które jednocześnie zawierają elementy do wykonywania złożonych obliczeń matematycznych. Bardzo często silniki gier zawierają w sobie silnik graficzny, który odpowiedzialny jest za renderowanie grafiki na ekranie urządzenia.

Rozdział 2

Biblioteka OpenGL

OpenGL *(ang. Open Graphics Library)* jest to wieloplatformowe *API*[1], które służy do renderowania grafiki 2D oraz 3D. Wspomniane API biblioteki OpenGL zostało zbudowane tak, iż wykorzystuje procesor graficzny (GPU) osiągając dzięki temu sprzętową akcelerację renderowania.

2.1 Historia powstania

Początek biblioteki OpenGL datuje się na tok 1982, w którym to firma Silicon Graphics Inc. rozpoczęła sprzedaż terminali graficznych używających własnościowego API zwanego wówczas *Iris Graphic Library*. W ciągu około dziesięciu lat SGI został liderem w grafice 3D dla stacji roboczych deklasując tym samym *PHIGS*[2]. Iris GL zostało uznane jako standard ze względu na łatwiejszą obsługę oraz natychmiastowy tryb renderowania w przeciwieństwie do swego konkurenta.

Konkurencyjne firmy, czyli między innymi Sun Microsystems, Hewlett-Packard oraz IBM wprowadzały na rynek własne pomysły poprzez rozszerzenia do biblioteki PHIGS powodując tym samym spadek zarobków dla SGI. Podjęto wówczas decyzję o uczynieniu Iris GL otwartym standardem, stąd nazwa OpenGL.

W 1995 roku firma Microsoft wprowadziła na rynek bibliotekę DirectX, która stała się głównym konkurentem dla OpenGL. Pod koniec roku 1997 rozpoczęła się wspólna inicjatywa

[1]API (ang. Application Programming Interface) - interfejs programowania aplikacji, ściśle określony zestaw reguł oraz ich opisów wraz z gotowymi narzędziami.

[2]PHIGS (ang. Programmer's Hierarchical Interactive Graphics System) - API służące do renderowania grafiki 3D uważane za standard w latach 1980-1990.

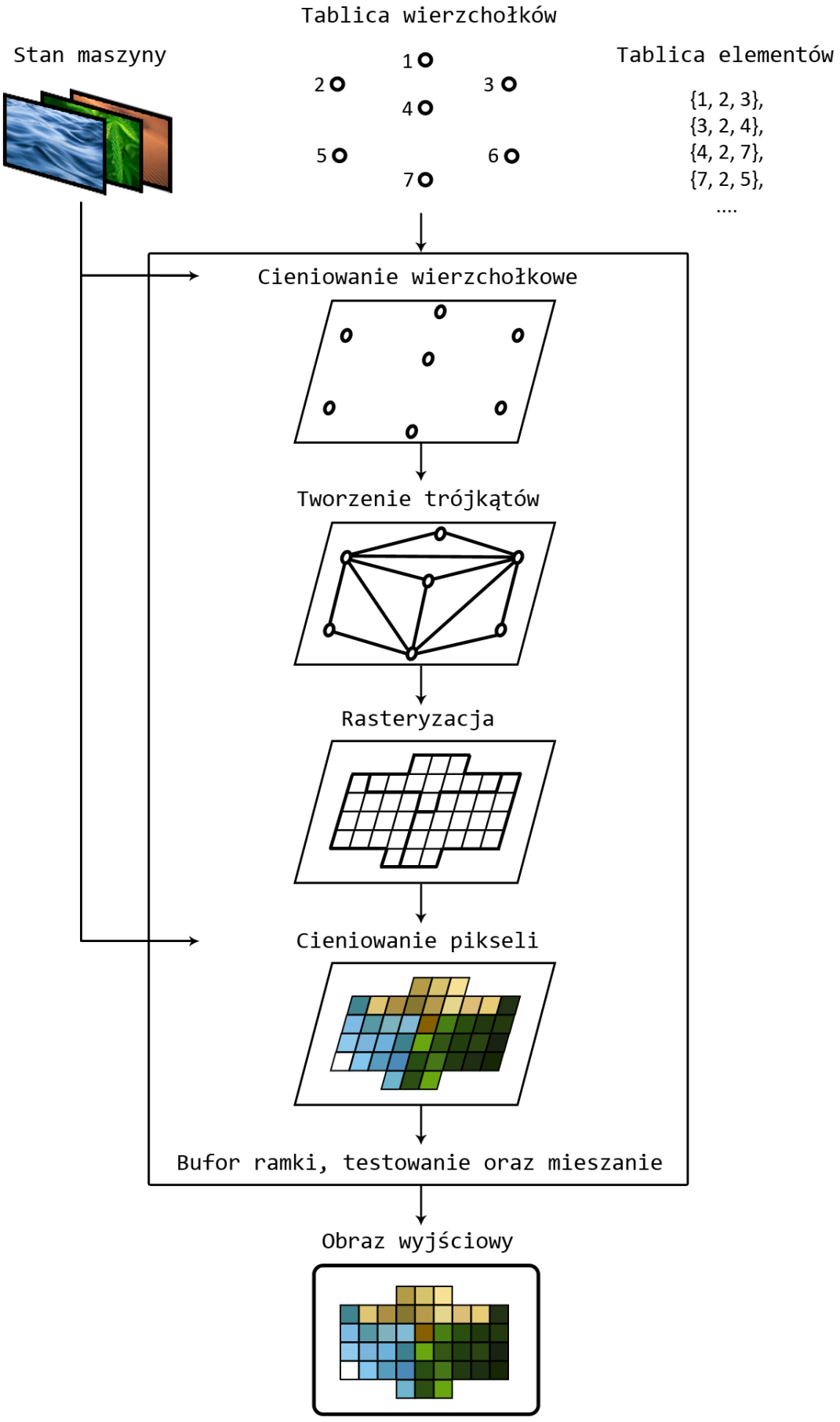
Microsoftu oraz SGI, która miała na celu ujednolicenie obu produktów tworząc tym samym *Fahrenheit graphics API*. W roku 1998 do projektu dołączyła firma Hewlett-Packard. Ostatecznie ze względów finansowych oraz strategicznych projekt został porzucony w roku 1999. W lipcu 2006 kontrolę standardu API OpenGL podjęła firma non-profit pod nazwą Khronos Group.

W chwili obecnej, implementacje OpenGL można znaleźć w praktycznie wszystkich nowoczesnych platformach.

2.2 Sposób działania

API biblioteki OpenGL jest zdefiniowane jako szereg funkcji składających się z około 250 podstawowych wywołań, umożliwiających budowanie złożonych trójwymiarowych scen z podstawowych figur geometrycznych. Biblioteka ta działa w architekturze klient-serwer. Klientem jest aplikacja wykorzystująca bibliotekę, która zleca operacje graficzne do wykonania. Natomiast serwerem jest aktualnie używana implementacja OpenGL. W przypadku aplikacji wykorzystującej OpenGL najczęściej znajdują się na tej samej maszynie. Nie jest to jednak warunek konieczny, ponieważ biblioteka została zaprojektowana tak, aby było możliwe wyświetlanie grafiki OpenGL na urządzeniu zdalnym.

Podstawową cechą OpenGL jest to, że jest on maszyną stanu. Na stan OpenGL w danym momencie składają się określone parametry oraz tryby działania. Konfiguracja trybów ma wpływ na rezultat renderingu. Raz ustawiony parametr lub tryb działania pozostaje zachowany, aż do następnej zmiany. Przykładami takich parametrów mogą być: kolor rysowania, aktualnie używana tekstura, sposób działania bufora Z, macierz na której wykonywane są aktualnie operacje oraz wiele innych. Przykładowymi metodami do zmiany trybów są m.in. `glEnable(...)` oraz `glDiable(...)`.



Rysunek 2.1 Uproszczony diagram działania biblioteki OpenGL.

Powyższy diagram prezentuje sposób renderowania grafiki z wykorzystaniem biblioteki OpenGL. Przejdźmy do opisu poszczególnych elementów:

- Tablica wierzchołków oraz elementów

  Praca renderera rozpoczyna się w momencie, gdy stworzony zostanie przynajmniej jeden bufor wierzchołków *(ang. vertex buffer)*, który wypełniany jest tablicami atrybutów *(ang. vertex attributes)*. Atrybuty te używane są jako dane wejściowe dla programów cieniujących *(ang. vertex shader)*. Powszechne atrybuty wierzchołków posiadają takie dane jak lokalizacja w przestrzeni 3D oraz jeden lub więcej zbiór koordynatów tekstury, które mapują współrzędne tekstur do współrzędnych wierzchołków. Zbiór buforów wierzchołków nazywany jest tablicą wierzchołków *(ang. vertex array)*. W momencie zgłoszenia wywołania renderera dostarczona zostaje dodatkowa tablica, zawierająca indeksy wierzchołków, które zostaną włączone do procesu renderowania. Kolejność indeksów wpływa na sposób w jaki wierzchołki będą łączone w trójkąty.

- Niezmienny stan oraz tekstury

  Praca renderera posiada niezmienny stan *(ang. uniform state)*, który zawiera zestaw danych służących tylko do odczytu potrzebnych na każdym etapie renderowania. Dzięki temu programy cieniujące mogą pobierać parametry obiektów bez jakichkolwiek zmian w tablicy wierzchołków lub fragmentów. Jednolity stan zawiera w sobie tekstury opisane w sposób jedno, dwu lub trójwymiarowej tablicy, które z kolei są wykorzystywane przez programy cieniujące.

- Cieniowanie wierzchołkowe

  Procesor graficzny rozpoczyna swoją pracę od wczytania wszystkich wybranych wierzchołków z tablicy. Następnie moduł cieniujący wierzchołki zmienia ich zestaw atrybutów w inny, taki który jest w stanie zrozumieć moduł rasteryzujący. Programy modułu cieniującego wierzchołkowe są w głównej mierze odpowiedzialne za obliczenie miejsca wierzchołka na ekranie. Mogą one jednak wygenerować dodatkowe dane wyjściowe takie jak kolor lub współrzędne tekstury.

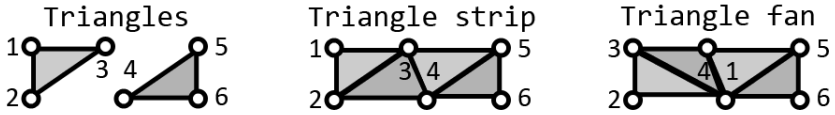
- Tworzenie trójkątów

  W obecnej fazie procesor graficzny łączy wierzchołki do formy trójkątów. Robi to poprzez tablicę elementów, która określa ich sposób grupowania w zbiory trójelementowe. Możemy wyróżnić kilka sposobów grupowania:

– pobieranie wszystkich trzech elementów jako odrębnych wierzchołków *(ang. triangles)*,

– tworzenie trójkątów używając dwóch poprzednich wierzchołków jako pierwsze dwa przy tworzeniu kolejnego *(ang. triangle strip)*,

– tworzenie trójkątów tworząc połączenia z jednego do dwóch kolejnych par wierzchołków *(ang. triangle fan)*.



Rysunek 2.2 Diagram obrazujący tworzenie trójkątów różnymi metodami.

- Rasteryzacja

  Moduł rasteryzujący pobiera każdy utworzony trójkąt, zaczepia go w przestrzeni, odrzuca elementy znajdujące się poza ekranem oraz tworzy pojedyncze fragmenty o rozmiarze piksela. Dane wygenerowane poprzez moduł cieniujący wierzchołki takie jak kolor są interpolowane względem całej powierzchni, która ma zostać zrasteryzowana. Dla przykładu, jeśli moduł cieniujący przypisze dowolny kolor do każdego wierzchołka wówczas moduł rasteryzujący wymiesza te kolory tworząc gradient.

Rysunek 2.3 Diagram obrazujący sposób rasteryzacji trójkąta o przypisanych kolorach wierzchołków.

- Cieniowanie pikseli

  Wygenerowane fragmenty poddawane są operacjom programu zwanego modułem cieniującym piksele *(ang. fragment shader lub pixel shader)*. Otrzymuje on dane modułu cieniowania wierzchołków oraz efekty interpolacji modułu rasteryzującego jako dane wejściowe. Dane zwrócone na wyjściu to kolor oraz wartość głębokości, które są rysowane na buforze ramki *(ang. framebuffer)*. Operacje cieniowania pikseli zawierają w sobie mapowanie tekstur oraz światła. Obywają się one niezależnie dla każdego piksela, stąd mogą zostać wykorzystane do tworzenia efektów specjalnych.

- Bufor ramki, testowanie oraz mieszanie

  Bufor ramki jest docelowym elementem obiektów w module renderującym. Należy wspomnieć, iż jest on czymś więcej niż dwuwymiarowym obrazem. W związku z posiadaniem przynajmniej jednego buforu koloru, bufor klatki może posiadać również bufor głębi lub bufor szablonowy. Oba te bufory dodatkowo filtrują fragmenty zanim zostaną ona narysowane w ramce. Bufor głębi odrzuca fragmenty, które znajdują się za elementami, które już zostały narysowane. Filtr szablonowy testuje czy rysowane kształty nie wychodzą poza widok ekranu, jeśli tak, są one usuwane. Fragmenty, które przetrwały przez oba filtry zostają poddane mieszaniu barw, operacjom głębi ostatecznie trafiając do bufora ramki.

Rozdział 3

Platforma Android

Android jest to platforma oparta na jądrze *Linux*[1]. W jej skład wchodzi system operacyjny oraz *SDK*[2], które umożliwia tworzenie aplikacji z użyciem języka Java. System ten przeznaczony jest głównie do użytku na urządzeniach mobilnych takich jak telefony komórkowe, tablety oraz netbooki.

3.1 Historia powstania

Historia systemu Android zapoczątkowana została w lipcu 2005 roku, kiedy to firma Google zakupiła niewielką firmę z Kalifornii - Android Inc.. Wówczas wiedziano jedynie, iż firma ta tworzyła oprogramowanie na potrzeby urządzeń mobilnych. Współtwórcy Android Inc. będąc już pod okiem firmy Google stworzyli nowatorski system operacyjny dla urządzeń mobilnych działający pod kontrolą jądra Linux.

Dnia 5 listopada 2007 roku założono Open Handset Alliance, konsorcjum w którego skład wchodzą między innymi takie firmy jak Google, HTC, Intel, Motorola, Qualcomm, T-Mobile, Sprint Nextel oraz NVIDIA. Ma ono na celu rozwój otwartych standardów dla telefonii mobilnej.

W dniu ogłoszenia powstania OHA zaprezentowano platformę Android. Niedługo potem, bo 12 listopada 2007 OHA opublikowało pierwszą wersję Android SDK dostępną do pobrania na oficjalnej stronie internetowej.

[1]Linux - uniksopodobny system operacyjny oparty o model wolnego oprogramowania.

[2]SDK (ang. Software Development Kit) - zestaw narzędzi dla programistów niezbędny w tworzeniu aplikacji

3.2 Dlaczego Android?

Do stworzenia silnika gier na środowisko Android przekonało mnie kilka czynników. Jak wspominałem jednym z nich był niewątpliwy wzrost popularności urządzeń z tym systemem w bardzo niewielkim czasie. Już wtedy można było powiedzieć, iż jest to wygodny system dla dużej liczby odbiorców. Na dowód tego poniżej przedstawiona została tabela sprzedaży urządzeń mobilnych z systemem Android na światowym rynku w latach 2012 oraz 2013.

System	1 kw. 2012	2 kw. 2012	3 kw. 2012	4 kw. 2012	1 kw. 2013
Android	36.40%	43.40%	75.00%	69.70%	74.40%
Pozostałe	63.60%	56.60%	25.00%	30.30%	25.60%

Źródło: *www.gartner.com*

Tabela 3.1 Sprzedaż urządzeń mobilnych z system Android w latach 2012 oraz 2013.

Dodatkowym bardzo istotnym kryterium na korzyść Androida był fakt, iż bardzo ciężko jest stworzyć grę na komputery stacjonarne, która mogłaby spełnić oczekiwania graczy. Nie wystarcza tutaj dobra grywalność lub wciągająca fabuła. W obecnych czasach większość graczy liczy głównie na szczegółowo odwzorowaną grafikę oraz fizykę, a także intuicyjne sterowanie postacią. *Android Market*[3] wyszło naprzeciw dając możliwość świeżego startu, pokazania nowych pomysłów, które mogą spodobać się użytkownikom.

3.3 Programowanie aplikacji mobilnej

Przed opisem gotowych funkcji silnika chciałbym przedstawić środowisko wraz z przykładowymi kodami źródłowymi. Dzięki temu będę mógł przedstawić sposób tworzenia aplikacji pod kontrolą systemu Android.

3.3.1 Podstawy

Pierwszą aplikacją z jaką zwykle możemy się spotkać w trakcie nauki nowego języka jest napisanie prostego programu, tak zwanego *"Hello World"*. Tworząc nowy projekt aplikacji dla systemu Android cała struktura projektu jest generowana automatycznie.

[3]Android Market (obecnie Google Play) - internetowy sklep firmy Google z aplikacjami, grami, muzyką oraz innymi dostępnymi multimediami.

Nazwa	Opis
assets/	Katalog z plikami użytkownika wykorzystywanymi w projekcie.
bin/	Katalog z plikami binarnymi (skompilowanymi plikami klas).
gen/	Katalog z wygenerowanymi materiałami.
libs/	Katalog z używanymi bibliotekami w projekcie.
res/	Katalog z ważnymi plikami zasobów (strings.ml, układy itp.).
src/	Katalog z kodami źródłowymi aplikacji.
AndroidManifest.xml	Plik konfiguracyjny z informacjami o projekcie.
build.properies	Plik właściwości budowania aplikacji.
build.xml	Standardowy plik Ant do sterownia budowy aplikacji.
project.properties	Posiada informacje odnośnie używanego SDK oraz bibliotek.

Tabela 3.2 Elementy generowane w czasie tworzenia projektu.

Automatycznie wygenerowany plik uruchamiający posiada nazwę *MainActivity.java*. Jego lokalizacja to *src/nazwa_pakietu/MainActivity.java*. Zawiera w sobie domyślne metody wraz automatycznie wygenerowanym kodem, który pojawił się dzięki dziedziczeniu z klasy `Activity`.

```
1 public class MainActivity extends Activity
2 {
3     @Override
4     protected void onCreate(Bundle savedInstanceState)
5     {
6       super.onCreate(savedInstanceState);
7       setContentView(R.layout.activity_main);
8     }
9 }
```

Kod źródłowy 3.1 Przykładowa aplikacja Hello World

Klasa ta odpowiedzialna jest za tworzenie okna aplikacji oraz za dostęp do innych podstawowych komponentów systemowych. Zazwyczaj jedna klasa aktywności reprezentuje jedno okno aplikacji.

W linii numer 4 utworzona została przeciążona metoda `onCreate(...)`, która wywoływana jest w momencie uruchamiania aplikacji. To tutaj powinna odbywać się inicjalizacja obiektów lub wczytywanie danych dla aplikacji Android.

Powyższy kod wykorzystuje również plik *XML*[4], który określa wygląd aplikacji. Zwiera

[4]XML (ang. Extensible Markup Language) - rozszerzalny język znaczników przeznaczony do reprezentowania dowolnych danych w sposób strukturalny

on takie dane jak użyte komponenty czy ustawienia szablonu.

```
1  <RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
2     xmlns:tools="http://schemas.android.com/tools"
3     android:layout_width="match_parent"
4     android:layout_height="match_parent"
5     android:paddingBottom="@dimen/activity_vertical_margin"
6     android:paddingLeft="@dimen/activity_horizontal_margin"
7     android:paddingRight="@dimen/activity_horizontal_margin"
8     android:paddingTop="@dimen/activity_vertical_margin"
9     tools:context=".MainActivity" >
10
11    <TextView
12       android:layout_width="wrap_content"
13       android:layout_height="wrap_content"
14       android:text="@string/hello_world" />
15
16 </RelativeLayout>
```

Kod źródłowy 3.2 Plik XML określający wygląd okna aplikacji

W liniach od 1 do 9 został określony typ układu *(ang. layout)*, marginesy oraz sposób dostosowania okna do rodzica, czyli w tym przypadku całego obszaru roboczego. W opisywanym fragmencie kodu (linie numer 11 - 14) występuje również kontrolka o nazwie `TextView`. Służy ona do wyświetlania tekstu na ekranie. W tym przypadku wartość text została określona przez string o nazwie `hello_world`. Wartości danych typu `string` zwykle zapisuje się w oddzielnym pliku *strings.xml*.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <resources>
3     <string name="app_name">CatEngine</string>
4     <string name="action_settings">Settings</string>
5     <string name="hello_world">Hello world!</string>
6 </resources>
```

Kod źródłowy 3.3 Plik XML z wartościami danych typu string.

3.3.2 Obsługa biblioteki OpenGL

Aplikacja opisana w poprzednim podrozdziale nie robi nic więcej, niż wyświetlanie tekstu przypisanego do kontrolki `TextView`. Przejdźmy zatem do przykładowego kodu, który dzięki rozszerzeniu o dodatkowe elementy posłużył za podstawę do budowy opisywanego silnika.

```
1 public class MainActivity extends Activity implements Renderer
2 {
3     private GLSurfaceView mGLView;
4
5     @Override
6     protected void onCreate(Bundle savedInstanceState)
7     {
```

```
8         super.onCreate(savedInstanceState);
9
10        mGLView = new GLSurfaceView(this);
11        mGLView.setRenderer(this);
12        setContentView(mGLView);
13      }
14
15      @Override
16      public void onSurfaceCreated(GL10 gl, EGLConfig config)
17      {
18        // TODO Auto-generated method stub
19      }
20
21      @Override
22      public void onDrawFrame(GL10 gl)
23      {
24        // TODO Auto-generated method stub
25      }
26
27      @Override
28      public void onSurfaceChanged(GL10 gl, int width, int height)
29      {
30        // TODO Auto-generated method stub
31      }
32  }
```

Kod źródłowy 3.4 Podstawowy kod obsługujący bibliotekę OpenGL.

Zmiany w stosunku do kodu źródłowego 3.3.2 to pojawienie się trzech nowych metod. Powstały one dzięki rozszerzeniu klasy o implementację interfejsu `Renderer`.

- `onSurfaceCreated(...)`
  Wywoływana jest w momencie, gdy obszar OpenGL jest tworzony lub odtwarzany. Wszystkie elementy graficzne korzystające z biblioteki graficznej powinny być wczytywane w tej metodzie.
- `onSurfaceChanged(...)`
  Wywoływana w momencie zmiany rozmiaru obszaru roboczego.
- `onDrawFrame(...)`
  Wywoływana w celu przerysowania klatki.

Dodatkowo zmieniony został kod w metodzie `onSurfaceCreated(...)`. To w niej bowiem przypisujemy informacje, z których wynika, że aplikacja ma korzystać z widoku biblioteki OpenGL wykorzystując wskazany obiekt zwany rendererem.

Istnieje wiele sposobów dostosowania środowiska oraz tworzonego okna do zewnętrznych zasobów sprzętowych. Ze względu na to, iż chciałem stworzyć silnik obsługujący grafikę dwuwymiarową w moim przypadku użyty został rzut ortogonalny na płaszczyznę. Upraszcza on bowiem przestrzeń sceny pomijając współrzędne osi Z.

```
1  @Override
2  public void onSurfaceCreated(GL10 gl, EGLConfig config)
3  {
4      gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
5      gl.glClearDepthf(1.0f);
6
7      gl.glEnable(GL10.GL_DEPTH_TEST);
8      gl.glEnable(GL10.GL_BLEND);
9      gl.glEnable(GL10.GL_DITHER);
10
11     gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
12     gl.glDepthFunc(GL10.GL_LEQUAL);
13     gl.glShadeModel(GL10.GL_SMOOTH);
14
15     gl.glHint(GL10.GL_LINE_SMOOTH_HINT, GL10.GL_NICEST);
16 }
17
18 @Override
19 public void onSurfaceChanged(GL10 gl, int width, int height)
20 {
21     gl.glViewport(0, 0, width, height);
22
23     gl.glMatrixMode(GL10.GL_PROJECTION);
24     gl.glLoadIdentity();
25
26     gl.glOrthof(0, width, 0, height, 0, Float.MAX_VALUE);
27     gl.glMatrixMode(GL10.GL_MODELVIEW);
28 }
29
30 @Override
31 public void onDrawFrame(GL10 gl)
32 {
33     // TODO Auto-generated method stub
34 }
```

Kod źródłowy 3.5 Podstawowy kod obsługujący bibliotekę OpenGL cz.2.

Powyższy kod przygotowuje środowisko OpenGL do rysowania podstawowych figur. Po jego uruchomieniu wyświetlony zostanie czarny ekran.

Metoda `onSurfaceCreated(...)` w liniach numer 1 i 2 czyści ekran oraz bufor głębokości. Następnie od lini 7 ustawia domyślne zmienne środowiska względem testu głębi oraz zmieszania pikseli. Ostatnia linia określa sposób wyświetlania elementów sceny. W tym przypadku została przypisana najlepsza jakość dla rysowanych linii.

Metoda `onSurfaceChanged(...)` określa rozmiar wyświetlanego okna na urządzeniu fizycznym. Dodatkowo ustala jak mają zostać wyświetlone elementy. W tym przypadku nie są one wyświetlane w sposób perspektywiczny (trójwymiarowy) lecz są rzutowane na przestrzeń dwuwymiarową.

Metoda `onDrawFrame(...)` obecnie nie posiada żadnych instrukcji. Chcąc wyświetlić dowolny obiekt kod renderujący należałoby umieścić w tym miejscu.

Tak przygotowany kod jest gotowy do wyświetlenia elementów graficznych na ekranie.

3.3.3 Wyświetlanie podstawowych obiektów

Chcąc przedstawić sposób wyświetlania podstawowych figur geometrycznych najwygodniej jest stworzyć dodatkową klasę, w której zawarte zostaną dodatkowe instrukcje.

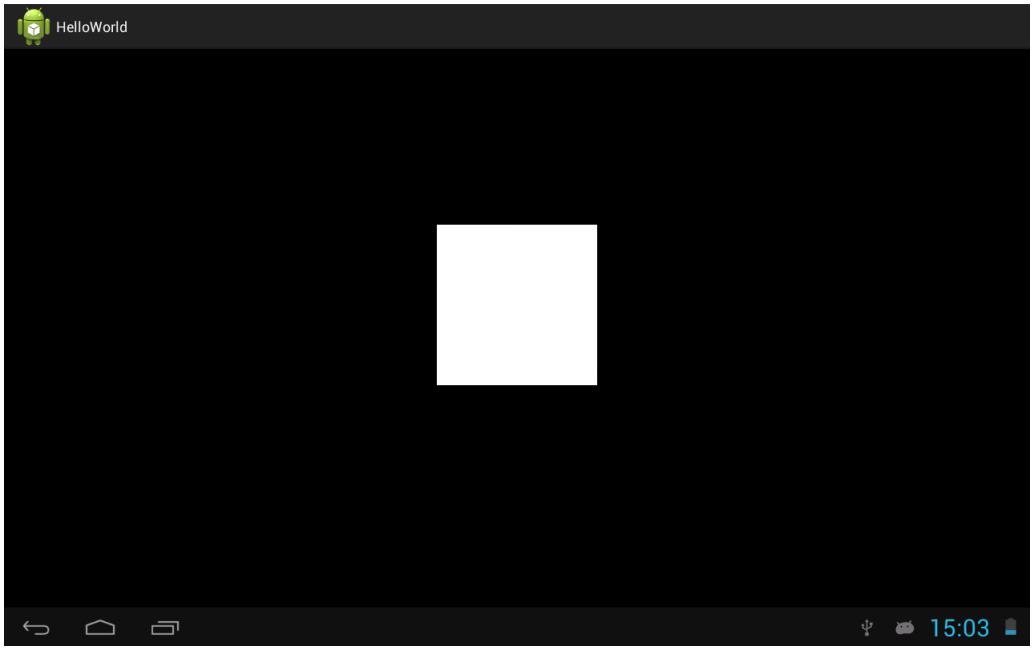
```
1  public class Rectangle
2  {
3      private float mColor[] = { 1f, 1f, 1f, 1f };
4
5      private float[] mVertices;
6      private int mVerticesCount;
7      private FloatBuffer mVertexBuffer;
8
9      public Rectangle(float x, float y, float width, float height)
10     {
11         mVertices = new float[] { x, y - height, // V1 - bottom left
12             x, y,                                // V2 - top left
13             x + width, y - height,               // V3 - bottom right
14             x + width, y                         // V4 - top right
15         };
16
17         mVerticesCount = mVertices.length / 2;
18         ByteBuffer byteBuffer = ByteBuffer.allocateDirect(mVertices.length * 4);
19         byteBuffer.order(ByteOrder.nativeOrder());
20
21         mVertexBuffer = byteBuffer.asFloatBuffer();
22         mVertexBuffer.put(mVertices);
23         mVertexBuffer.position(0);
24     }
25
26     public void draw(GL10 gl)
27     {
28       gl.glPushMatrix();
29
30         gl.glEnable(GL10.GL_BLEND);
31         gl.glEnable(GL10.GL_ALPHA_BITS);
32
33         gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
34         gl.glColor4f(mColor[0], mColor[1], mColor[2], mColor[3]);
35         gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
36
37         gl.glFrontFace(GL10.GL_CW);
38         gl.glVertexPointer(2, GL10.GL_FLOAT, 0, this.mVertexBuffer);
39         gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, this.mVerticesCount);
40
41         gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
42         gl.glDisable(GL10.GL_ALPHA_BITS);
43         gl.glDisable(GL10.GL_BLEND);
44
45         gl.glPopMatrix();
46     }
47 }
```

Kod źródłowy 3.6 Klasa Rectangle przygotowana dla środowiska OpenGL

Konstruktor klasy `Rectangle` jako parametry przyjmuje punkt, w którym ma się znajdować prostokąt oraz jego wielkość. W trakcie jego wywołania następuje stworzenie tablicy wierzchołków i wypełnienie bufora wykorzystywanego przy rysowaniu.

Metoda `draw(...)` aktywuje opcje potrzebne do renderowania danego obiektu poprzez `gl.glEnable(...)` określając sposób mieszania pikseli, kolor, metodę renderowania oraz rysuje obiekt z wykorzystaniem wcześniej przygotowanego bufora. Ostatecznie wyłącza opcje środowiska poprzez `glDisable(...)`, które zostały aktywowane na początku metody.

Chcąc wyświetlić dowolny obiekt klasy `Rectangle` wystarczy go utworzyć w metodzie `onSurfaceCreated(...)` po wczytaniu wszystkich niezbędnych bibliotek, managerów i zmiennych. Następnie wewnątrz metody `onDrawFrame(...)` wywołujemy funkcję rysowania danego obiektu.



Rysunek 3.1 Efekt wyświetlania obiektu klasy Rectangle.

3.3.4 Teksturowanie obiektów

Posiadając wiedzę dotyczącą wyświetlania jednej z podstawowych figur geometrycznych jaką jest prostokąt trzeba wspomnieć o możliwości teksturowania obiektów.

Należy pamiętać o tym, iż biblioteka OpenGL do swojej pamięci jako tekstury przyjmuje pliki graficzne, których szerokość oraz wysokość jest podzielna bez reszty przez liczbę 2. Przez tego typu założenia należy wczytywać obrazy o stosownych rozmiarach. Chcąc ułatwić pracę użytkownikowi możemy stworzyć metodę, która wczyta plik graficzny oraz dostosuje go (w razie konieczności) do wymienionych standardów.

```
1 AssetManager mAssetManager;
```

```
2  GL10 mGL;
3
4  public Texture loadTexture(String pathName)
5  {
6     Bitmap loadedBitmap = getBitmapFromAsset(pathName);
7
8     if (loadedBitmap == null)
9     {
10       return null;
11    }
12
13    return new Texture(mGL, loadedBitmap);
14 }
15
16 private Bitmap getBitmapFromAsset(String pathName)
17 {
18    try
19    {
20        InputStream istr = mAssetManager.open(pathName);
21        Bitmap bitmap = BitmapFactory.decodeStream(istr);
22
23        return bitmap;
24    }
25    catch (IOException e)
26    {
27        e.printStackTrace();
28    }
29
30    return null;
31 }
```

Kod źródłowy 3.7 Metoda zwracająca teksturę utworzoną z wczytanej bitmapy.

Powyższy kod przedstawia metodę służącą do utworzenia tekstury z pliku o ścieżce podanej jako parametr.

Metoda `loadTexture(...)` zwraca teksturę utworzoną z bitmapy wczytanej przez `getBitmapFromAsset(...)`.

Metoda `getBitmapFromAsset(...)` wczytuje dowolny obraz z katalogu *assets/* znajdującego się w projekcie.

Zakładam, iż zmienne `mGL` oraz `mAssetManager` zostały wcześniej przypisane w kodzie na początku metody `onSurfaceCreated(...)`.

```
1  public void onSurfaceCreated(GL10 gl, EGLConfig config)
2  {
3     mGL = gl;
4     mAssetManager = getAssets();
5
6     // Dalsza czesc kodu
7  }
```

Kod źródłowy 3.8 Sposób inicjalizacji zmiennych mGL oraz mAssetManager.

Przejdźmy do klasy `Texture`, której konstruktor został wykorzystany do zwrócenia zmiennej w metodzie `loadTexture(...)`.

```
1  public class Texture
2  {
3     private int[] mTexturePointer = new int[1];
4      private FloatBuffer mTextureBuffer;
5      private ByteBuffer mByteBuffer;
6
7      public int imageWidth, imageHeight, textureWidth, textureHeight;
8
9      public Texture(GL10 gl, Bitmap bitmap)
10     {
11       float regionCoords[] = new float[] {
12          0.0f, 1.0f,
13          0.0f, 0.0f,
14          1.0f, 1.0f,
15          1.0f, 0.0f
16       };
17
18       imageWidth = textureWidth = bitmap.getWidth();
19       imageHeight = textureHeight = bitmap.getHeight();
20
21       if (isPowerOfTwo(textureWidth))
22       {
23          textureWidth = nextPowerOfTwo(textureWidth);
24       }
25
26       if (isPowerOfTwo(textureHeight))
27       {
28          textureHeight = nextPowerOfTwo(textureHeight);
29       }
30
31       if (textureWidth != imageWidth || textureHeight != imageHeight)
32       {
33          Bitmap tmpBitmap = Bitmap.createBitmap(textureWidth, textureHeight, Config.
                ARGB_8888);
34          Canvas canvas = new Canvas(tmpBitmap);
35          canvas.drawBitmap(bitmap, new Matrix(), null);
36
37          regionCoords[1] = bitmap.getHeight() / (float) textureHeight;
38          regionCoords[5] = bitmap.getHeight() / (float) textureHeight;
39          regionCoords[4] = bitmap.getWidth() / (float) textureWidth;
40          regionCoords[6] = bitmap.getWidth() / (float) textureWidth;
41
42          bitmap = tmpBitmap;
43       }
44
45       gl.glGenTextures(1, mTexturePointer, 0);
46       gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexturePointer[0]);
47       gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, GL10.
            GL_NEAREST);
48       gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, GL10.
            GL_LINEAR);
49
50       GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
51
52       mByteBuffer = ByteBuffer.allocateDirect(32);
53       mByteBuffer.order(ByteOrder.nativeOrder());
54       mTextureBuffer = mByteBuffer.asFloatBuffer();
55       mTextureBuffer.put(regionCoords);
56       mTextureBuffer.position(0);
57
58       bitmap.recycle();
59     }
60
61     public int getTexturePointer()
62     {
63       return mTexturePointer[0];
64     }
```

```
65
66      public FloatBuffer getTextureBuffer()
67      {
68        return mTextureBuffer;
69      }
70
71      private int nextPowerOfTwo(int value){ ... }
72      private boolean isPowerOfTwo(int value) { ... }
73  }
```

Kod źródłowy 3.9 Podstawowa klasa Texture.

Konstruktor klasy `Texture` tworzy tablicę z domyślnymi danymi względem której tekstura będzie mapowana do wierzchołków dowolnego obiektu. W liniach numer 18 i 19 wczytuje rozmiar wczytanej bitmapy. Jeśli szerokość lub wysokość nie jest potęgą liczby 2 znajdujemy najbliższą taką wartość, która jest większa od obecnej. Sprawdzamy czy wielkość obrazu oraz wielkość tekstury jest równa. Jeśli nie to tworzymy nową bitmapę o rozmiarach `textureWidth` oraz `textureHeight`. Do nowo powstałej bitmapy przerysowujemy wczytany obraz i modyfikujemy tablicę `regionCoords` w taki sposób, aby jej punkty określały współrzędne wierzchołków obrazu na teksturze. Ostatecznie bindujemy bitmapę jako teksturę i tworzymy bufor dla wcześniej stworzonej tablicy.

Metoda `getTexturePointer()` zwraca wskaźnik na teksturę załadowaną do środowiska OpenGL.

Metoda `getTextureBuffer()` zwraca bufor tekstury, który jest tworzony na podstawie `regionCoords`.

Posiadając obiekt klasy `Texture` brakuje jedynie sposobu wyświetlania wczytanej grafiki. Analogicznie jak wcześniej tworzymy dodatkową klasę, która będzie za to odpowiedzialna.

```
1  public class Sprite extends Rectangle
2  {
3      private Texture mTexture;
4
5      public Sprite(float x, float y, Texture texture)
6      {
7        super(x, y, texture.imageWidth, texture.imageHeight);
8        mTexture = texture;
9      }
10
11     public void draw(GL10 gl)
12     {
13       gl.glPushMatrix();
14
15       gl.glEnable(GL10.GL_BLEND);
16       gl.glEnable(GL10.GL_ALPHA_BITS);
17       gl.glEnable(GL10.GL_TEXTURE_2D);
18
19       gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
20       gl.glColor4f(mColor[0], mColor[1], mColor[2], mColor[3]);
21       gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexture.getTexturePointer());
22       gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
```

```
23        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
24
25        gl.glFrontFace(GL10.GL_CW);
26        gl.glVertexPointer(2, GL10.GL_FLOAT, 0, mVertexBuffer);
27        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTexture.getTextureBuffer());
28        gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, mVerticesCount);
29
30        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
31        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
32        gl.glDisable(GL10.GL_TEXTURE_2D);
33        gl.glDisable(GL10.GL_ALPHA_BITS);
34        gl.glDisable(GL10.GL_BLEND);
35
36        gl.glPopMatrix();
37      }
38  }
```

Kod źródłowy 3.10 Podstawowa klasa Sprite.

Klasa `Sprite` powstała poprzez dziedziczenie klasy `Rectangle`. Ze względu na powtarzające się elementy kodu jest to znaczne ułatwienie. Podstawowe pola zostały rozszerzone o zmienną `mTexture`, która będzie przechowywać wcześniej załadowaną teksturę wykorzystywaną przy rysowaniu obiektu.

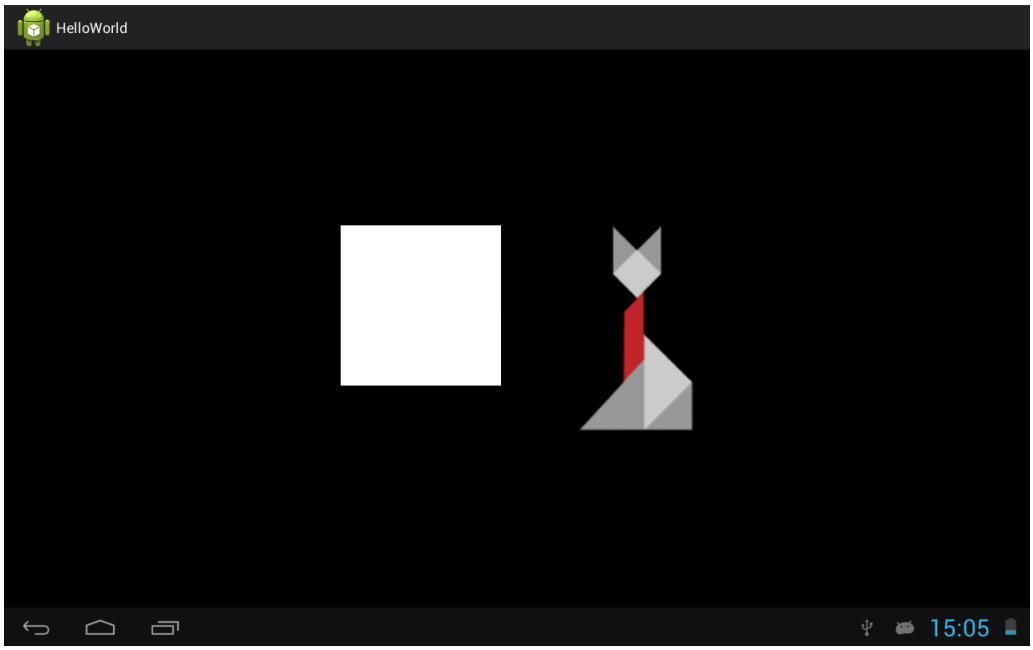
Konstruktor klasy `Sprite` wywołuje metodę `super(...)`, która tworzy obiekt z klasy nadrzędnej. Następnie zapamiętuje teksturę.

Metoda `draw(...)` została poddana modyfikacjom dzięki którym biblioteka OpenGL wyświetla dwuwymiarowy obraz na obszarze prostokąta. W odróżnieniu do metody rysowania z klasy nadrzędnej zostały dodane następujące linie:

- `gl.glEnable(GL10.GL_TEXTURE_2D);`
  Informuje środowisko o rozpoczęciu prac z użyciem tekstur.
- `gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexture.getTexturePointer());`
  Binduje teksturę poprzez wskaźnik pobrany z tablicy.
- `gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);`
  Przypisuje stan, w którym środowisko OpenGL jest w stanie wyświetlić teksturę za pomocą współrzędnych.
- `gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTexture.getTextureBuffer());`
  Przypisuje współrzędne wierzchołków tekstury względem podanego bufora.
- `gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);`
  Dezaktywuje stan rysowania tekstur.

- `gl.glDisable(GL10.GL_TEXTURE_2D));`
  Informuje środowisko o zakończeniu prac z użyciem tekstur.

Chcąc wyświetlić obiekt graficzny w scenie należy załadować teksturę poprzez metodę `loadTexture(...)` i przekazać ją jako parametr wywołania konstruktora klasy `Sprite`.



Rysunek 3.2 Efekt wyświetlania obiektów klasy Rectangle oraz Sprite.

Kod, który został opisany powyżej posłużył jako punkt wyjściowy do stworzenia silnika gier. Zawiera on bowiem podstawowe oraz jednocześnie najważniejsze elementy obsługi środowiska OpenGL, które można rozszerzyć o dodatkowe metody.

Rozdział 4

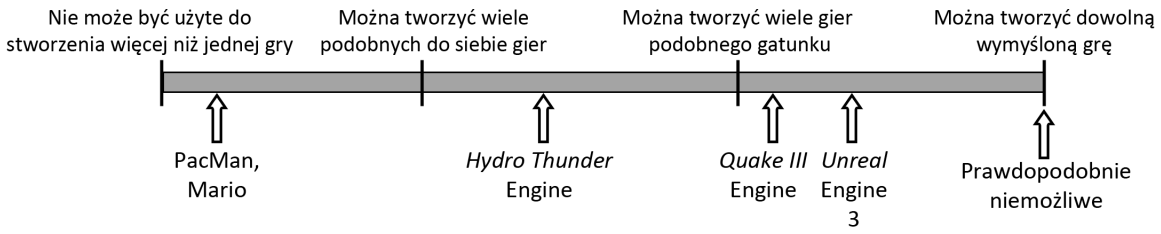
Silnik gier

Zagadnienie "silnika gier" powstało w połowie lat 1990 w odniesieniu do gier typu *first-person shooter* takich jak *Doom* wydanej przez *id Software*. *Doom* został zaprojektowany tak, że w widoczny sposób oddzielał część składowników oprogramowania (np. trójwymiarowy render środowiska, system detekcji kolizji lub system audio) oraz pozostałe elementy takie jak zbiór zasobów, świat gry, czy też jej zasady, które składały się na grywalność całego produktu. Wartość tego typu separacji elementów miała znaczenie w momencie, gdy producenci gier zaczęli licencjonować swoje produkty oraz zmieniać je w nowy produkt poprzez zmianę grafiki, świata przedstawionego czy zasad gry w minimalnym nakładem siły we wcześniej stworzonym "silniku". Tego typu zachowanie rozpoczęło działalność małych firm, które tworzyły nowe gry na podstawie starych. W związku z tym, pod koniec lat 1990 powstały takie gry jak *Unreal* lub *Quake III Arena*, które były otwarte na tego typu modyfikacje.

Granica między grą oraz silnikiem gry wydaje się być mało precyzyjna. Niektóre projekty tworzą wyraźną granicę, inne nawet nie próbują oddzielić tych zagadnień. W jednej grze kod renderujący może wiedzieć w jaki sposób wyświetlić dany obiekt. Inna gra może posiadać elementy ogólnego przeznaczenia takie jak materiały czy programy cieniujące, dzięki którym zostanie stworzony ten sam obiekt. W związku z tym, żadna z grup projektanckich nie podejmuje się perfekcyjnego oddzielenia gry oraz silnika, ze względu na to, iż obie definicje często przenikają się.

Elementem, który odróżnia silnik od gry jest prawdopodobnie architektura oparta na danych *(ang. data-driven architecture)*. W momencie, gdy gra zawiera niezmienną implementację logiki, czy też kod, który renderuje tylko konkretne obiekty, wówczas niemal niemożliwe staje się ponowne użycie produktu w innym celu. W tym momencie, możemy określić, iż "silnikiem gier" jest oprogramowanie, które jest w pewnym stopniu rozszerzalne i może posłużyć

za podstawę do tworzenia różnych gier bez dużej ilości modyfikacji.

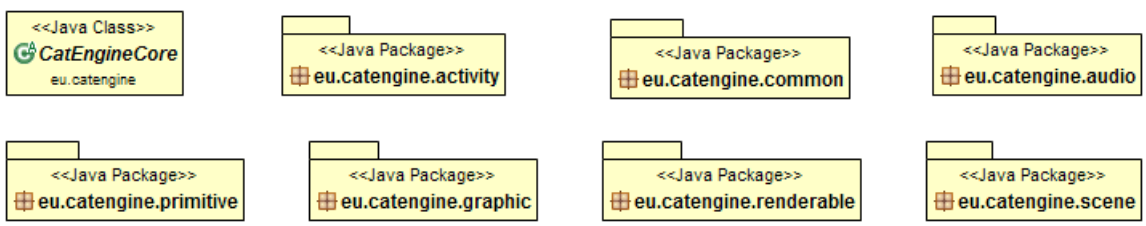


Rysunek 4.1 Diagram zależności między silnikiem i jego produktami.

Powyżej został przedstawiony diagram, który prezentuje pewne zależności między znanymi grami oraz silnikami gier. W tym przypadku, podobnie jak w życiu nie wszystko jest czarno-białe, występują bowiem odcienie szarości. Wnioski jakie można wysnuć z powyższego diagramu są takie, że nie da się stworzyć idealnego silnika do wszystkich zastosowań. Każde tego typu oprogramowanie jest w pewien sposób ograniczone. Wynika z tego to, że do różnego typu gier należy używać różnego typu silników.

4.1 Klasy oraz przykłady użycia

W celu ułatwienia pracy wszystkie elementy silnika zostały podzielone na tak zwane pakiety. Pakiety podobnie jak katalogi mają strukturę hierarchiczną. Oznacza to, iż każdy z pakietów może zawierać w sobie kolejne pakiety oraz dowolną ilość klas czy interfejsów. Nazwy pakietów zwyczajowo pisze się małymi literami. Kolejne poziomy zagnieżdżenia oddziela się od siebie kropkami, analogicznie do tego jak nazwy katalogów w ścieżce dostępu do pliku oddziela się od siebie ukośnikami. Firma Sun często zaleca, by w nazewnictwie pakietu stosować odwróconą nazwę domeny internetowej.



Rysunek 4.2 Diagram przedstawiający elementy pakietu eu.catengine.

4.1.1 CatEngineCore

CatEngineCore to podstawowa klasa silnika rozszerzająca klasę `Activity` umożliwiającą uruchomienie aplikacji w systemie Android. Posiada w sobie obiekty klas pomocniczych służących do organizacji pracy poszczególnych elementów silnika.

Chcąc stworzyć nową grę należy rozszerzyć domyślnie stworzone activity poprzez tą klasę. Po tej operacji należy uzupełnić kod metod abstrakcyjnych wygenerowanych po dziedziczeniu klasy. Metody te wywoływane są automatycznie w trakcie pracy silnika.

Ważne elementy

- Dostosowanie okna gry względem wielkości ekranu urządzenia.

  Użytkownik określa rozmiar okna gry poprzez przypisanie rozdzielczości ekranu jaką chciałby zastosować. Jak powszechnie wiadomo różne urządzenia posiadają różne wielkości oraz rozdzielczości ekranów. Istnieje kilka sposobów, z których można skorzystać, aby dostosować produkt wyjściowy.

  – Tworzenie kilku aplikacji, każdej pod inną rozdzielczość - czasochłonny sposób, tworzenia aplikacji lecz najbardziej wydajny dla większości urządzeń. W tego typu rozwiązaniach stosuje się różnego rozmiaru tekstury w zależności od możliwości ekranu. Nie wystąpi tutaj sytuacja w której zawsze wczytuje się duże tekstury niezależnie od rozdzielczości. Przykładowo pakiet tekstur dla urządzenia o rozdzielczości ekranu 800x480 pikseli powinien być około osiem razy mniejszy od pakietu dla rozdzielczości 2048x1536 pikseli.

  – Rozciąganie okna aplikacji do rozmiarów ekranu urządzenia bez zachowania proporcji - szybka, lecz mało efektywna metoda. Rozciągnięty obraz jest mało czytelny dla użytkownika, a co za tym idzie szybko go zirytuje.

  – Rozciągnięcie okna aplikacji do rozmiarów ekranu urządzenia z zachowaniem proporcji - metoda równie szybka jak poprzednia, jednak z lepszym efektem wizualnym dla gracza. W przypadku, gdy dana rozdzielczość będzie inna niż rozdzielczość ekranu przy oknie aplikacji mogą pojawić się czarne paski po bokach ekranu.

  Żadna z tych metod nie jest idealna. W stworzonym silniku została zaimplementowana ostatnia z opisywanych. Należy jednak wziąć pod uwagę to, iż programista może stworzyć kilka aplikacji obsługujących różne jakości. Najlepszym rozwiązaniem jest

stworzenie czterech wersji aplikacji dla ekranów o rozdzielczości określanej jako niskiej, średniej, wysokiej oraz HD. Tego typu aplikacje należy tworzyć na najbardziej popularne rozdzielczości z danej grupy. Dzięki temu chociaż część użytkowników nie będzie posiadać czarnych pasków po bokach swojego ekranu.

```
1  @Override
2  public void onSurfaceChanged(GL10 gl, int width, int height)
3  {
4     mScreenManager.updateViewportSize(width, height);
5     gl.glViewport(mScreenManager.getOffsetX(), mScreenManager.getOffsetY(),
           mScreenManager.getViewportWidth(), mScreenManager.getViewportHeight());
6
7     gl.glMatrixMode(GL10.GL_PROJECTION);
8     gl.glLoadIdentity();
9
10    gl.glOrthof(0, mScreenManager.getScreenWidth(), 0, mScreenManager.
          getScreenHeight(), -1.0f, 1.0f);
11    gl.glMatrixMode(GL10.GL_MODELVIEW);
12 }
```

Kod źródłowy 4.1 Metoda określająca rozmiar okna aplikacji.

W linii numer 4 została wykorzystana metoda `updateViewportSize(...)`, która oblicza wszystkie wartości służące do ustalenia rozmiaru oraz pozycji okna aplikacji. Metoda ta zostanie omówiona szczegółowo przy klasie `ScreenManager`.

Następnie metoda `gl.glViewport(...)` ustala odległość okna od lewego dolnego rogu ekranu urządzenia oraz jego wielkość. Poprzez `gl.glMatrixMode(GL10.GL_PROJECTION)` ustalamy macierz projekcji. Po załadowaniu macierzy jednostkowej ustalamy rzut ortogonalny o obliczonych rozmiarach. Ostatecznie powracamy do domyślnej macierzy widoku modelu.

Przykłady użycia

- Tworzenie oraz dodawanie domyślnej sceny.

```
1  public class MainActivity extends CatEngineCore
2  {
3      @Override
4      protected void loadSettings()
5      {
6         setFPS(50);
7        getScreenManager().setScreenResolution(1280, 754);
8      }
9
10     @Override
11     protected void loadResources()
12     {
13       getFontManager().loadFont("Verdana.ttf", 12);
14       getAudioManager().loadAudio("demo-audio.mp3");
15
```

```
16        Texture texture = getTextureManager().loadTexture("demo-texture.jpg");
17        getAnimationManager().loadAnimation("demo-animation", texture, new Size
              (100, 100), 10, 500);
18      }
19
20      @Override
21      protected void createScenes()
22      {
23        getSceneManager().addScene(new DemoScene("Demo"));
24      }
25
26      @Override
27      protected void onEnginePause()
28      {
29      }
30
31      @Override
32      protected void onEngineResume()
33      {
34      }
35  }
```

Kod źródłowy 4.2 Przykład użycia klasy CatEngineCore.

`loadSettings()` - służy do określenia ustawień silnika. W tym przypadku została określona ilość pożądanych klatek na sekundę oraz rozdzielczość okna gry.

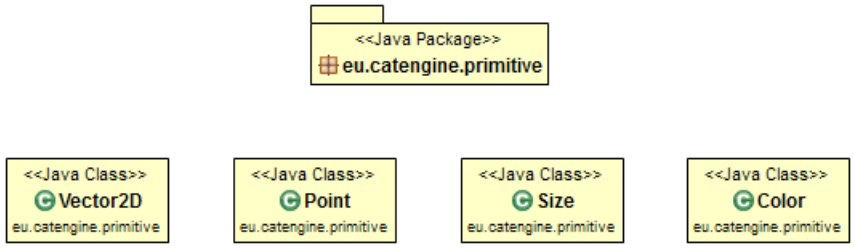
`loadResources()` - służy do wczytania wszystkich zewnętrznych elementów takich jak tekstury, dźwięki, animacje. W tym przypadku została wczytana czcionka *Verdana* o wielkości 12 punktów, plik muzyczny *demo-audio.mp3*, tekstura *demo-texture.jpg* oraz animacja o nazwie *demo-animation*.

`createScenes()` - służy do utworzenia obiektów scen i dodanie ich do managera sceny. Należy dodać przynajmniej jedną scenę, aby silnik mógł zostać uruchomiony. Pierwsza dodawana scena automatycznie staje się aktywna. Oznacza to, że będzie ona wyświetlana na ekranie jako pierwsza.

`onEnginePause()` oraz `onEngineResume()` - wywołują się w momencie przejścia aplikacji w stan pauzy lub odwrotnie.

4.1.2 Typy prymitywne

Typy prymitywne z pakietu `eu.catengine.primitive` charakteryzują się publicznym dostępem do swoich zmiennych. Ze względu na ilość wykorzystywanych obiektów tego typu korzystanie z funkcji `get()` oraz `set(...)` byłoby zbyt czasochłonne, a co za tym idzie wpływałoby na pracę silnika. Można zatem powiedzieć, iż jest to zabieg optymalizacyjny.



Rysunek 4.3 Diagram przedstawiający elementy pakietu eu.catengine.primitive.

4.1.2.1 Color

Klasa `Color` służy do przechowywania danych oraz metod koloru przypisywanego do różnego rodzaju obiektów. Posiada cztery składowe pracujące w zakresie $[0, 1]$:

- red - określa nasycenie barwy czerwonej,
- green - określa nasycenie barwy zielonej,
- blue - określa nasycenie barwy niebieskiej,
- alpha - określa stopień przezroczystości wyświetlanego obiektu.

Przykłady użycia

- Określenie koloru sceny oraz jej obiektów.

```
1    setBackgroundColor(new Color(0.5f, 0.5f, 0.5f, 1.0f));
2
3    Shape rectangle = getSceneTree().getObject("rectangle");
4    rectangle.setColor(new Color(1.0f, 0.0f, 0.0f, 0.5f));
```

Kod źródłowy 4.3 Przykład użycia klasy Color.

Przedstawiony fragment kodu zmienia tło sceny do koloru szarego. Dodatkowo kolor obiektu rectangle został zmieniony do `#FF0000` posiadając przy tym 50% przezroczystości. Istnieje możliwość kolorowania dowolnych elementów wywodzących się z klasy `Shape`.

4.1.2.2 Point

Klasa `Point` służy do przechowywania danych oraz metod punktu zaczepionego w przestrzeni 2D. Dziedziczy po klasie `PointF` z pakietu `android.graphics`. Posiada dwie składowe pracujące w zakresie liczb typu `float`:

- x - określa współrzędną osi X,
- y - określa współrzędną osi Y.

Klasa ta posiada metody służące do dodawania, odejmowania, mnożenia, dzielenia przez skalar lub obiekt tej samej klasy. Dodatkowo można obliczyć odległość między dwoma dowolnymi punktami.

Przykłady użycia

- Określenie pozycji kamery oraz obiektów sceny.

```
1    getSceneTree().addObject(new Rectangle(new Point(500, 500), new Size(128,
         128)));
2
3    getCamera().moveTo(new Point(100, 100));
```

Kod źródłowy 4.4 Przykład użycia klasy Point.

Przedstawiony fragment kodu tworzy obiekt klasy `Rectangle` w punkcie (500, 500) oraz przesuwa kamerę do punktu (100, 100).

4.1.2.3 Size

Klasa `Size` służy do przechowywania danych oraz metody rozmiaru różnego typu obiektów dwuwymiarowych. Posiada dwie składowe pracujące w zakresie liczb typu `float`:

- width - określa szerokość,
- height - określa wysokość.

Klasa ta posiada metody służące do dodawania, odejmowania, mnożenia, dzielenia przez skalar lub obiekt tej samej klasy.

Przykłady użycia

- Odczytywanie rozmiaru obiektu sceny.

```
1    Size size = getTextureManager().get("demo-texture").getSize();
2
3    text.setText("Rozmiar␣tekstury␣'demo-texture'␣to␣" + size.toString());
```

Kod źródłowy 4.5 Przykład użycia klasy Size.

Przedstawiony fragment kodu wyświetla na ekranie rozmiar tekstury *demo-texture*.

4.1.2.4 Vector

Klasa `Vector` służy do przechowywania danych dotyczących wektora w przestrzeni dwuwymiarowej. Posiada dwie składowe pracujące w zakresie liczb typu `float`:

- x - określa punkt zaczepienia wektora na osi X,
- y - określa punkt zaczepienia wektora na osi Y.

Klasa ta posiada metody służące do dodawania, odejmowania, mnożenia, dzielenia przez skalar lub obiekt tej samej klasy. Dodatkowo możemy znormalizować dany wektor oraz obliczyć długość lub iloczyn skalarny między dwoma wektorami.

Przykłady użycia

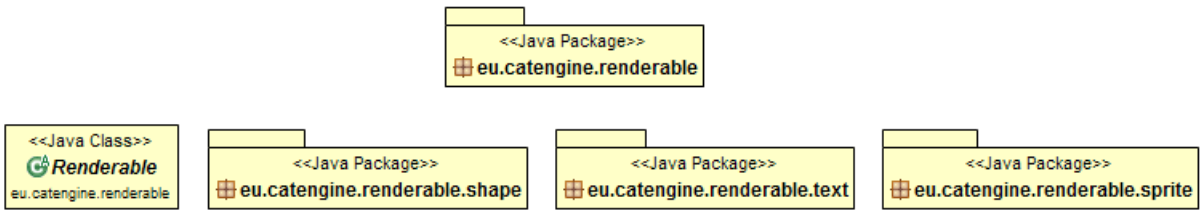
- Przesuwanie kamery oraz obiektów sceny o dany wektor.

```
1    getCamera().moveBy(new Vector(150, 20));
2
3    getSceneTree().getObject("rectangle").moveBy(new Vector(300, 100));
```

Kod źródłowy 4.6 Przykład użycia klasy Vector.

Przedstawiony fragment kodu przesuwa kamerę o wektor (150, 20) oraz obiekt o nazwie `rectangle` o wektor (300, 100).

4.1.3 Typy renderowane



Rysunek 4.4 Diagram przedstawiający elementy pakietu eu.catengine.renderable.

Klasy zawarte w pakiecie `eu.catengine.renderable` jak sama nazwa wskazuje są możliwe do wyrenderowania, czyli wyświetlania na ekranie. Dzielą się na kształty, elementy graficzne oraz tekstowe.

4.1.3.1 Renderable

Klasa `Renderable` przechowuje dane oraz metody niezbędne do wyświetlania podstawowych elementów sceny. Jest to klasa abstrakcyjna, służy za prototyp do tworzenia obiektów sceny, które nie wymagają wielu dodatkowych metod.

Przykłady użycia

- Tworzenie prostych obiektów sceny.

```
1  public class ParticleSystem extends Renderable
2  {
3      Random mRandom;
4      Rectangle[] mParticleArray;
5      Vector2D[] mVectorArray;
6      Point mLocation;
7
8      public ParticleSystem(int count, Point location)
9      {
10       mLocation = location;
11
12       mRandom = new Random();
13       mParticleArray = new Rectangle[count];
14       mVectorArray = new Vector2D[count];
15
16       for(int i=0; i<mParticleArray.length; i++)
17       {
18          Point randomPoint = new Point(location).add(mRandom.nextInt(10) - 5,
                mRandom.nextInt(10) - 5);
19          mParticleArray[i] = new Rectangle(randomPoint, new Size(10, 10));
20          mVectorArray[i] = new Vector2D(location, randomPoint);
21          mParticleArray[i].setColor((float) mRandom.nextDouble(), (float) mRandom
                .nextDouble(), (float) mRandom.nextDouble());
22       }
23
24       setUpdateNeeded(true);
25     }
26
27     @Override
28     public void draw(GL10 gl)
29     {
30       for(int i=0; i<mParticleArray.length; i++)
31       {
32          mParticleArray[i].draw(gl);
33       }
34     }
35
36     @Override
37     public void update(long elapsedTime)
38    {
39       for(int i=0; i<mParticleArray.length; i++)
40       {
41          mParticleArray[i].moveBy(mVectorArray[i].divide(elapsedTime));
42          mParticleArray[i].rotateByAtCenterPoint(10.0f);
43          mVectorArray[i].y -= 1.0f;
44       }
45     }
```

```
46 }
```

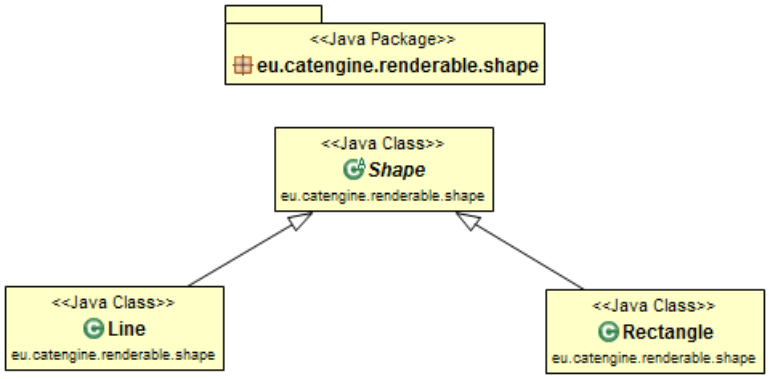
Kod źródłowy 4.7 Przykład użycia klasy Renderable.

Przedstawiony fragment kodu tworzy klasę `ParticleSystem`. W tym przypadku wyświetla na ekranie prosty efekt fajerwerków o ilości świecących punktów równych `count` w początkowym `location`.

Zasada działania tego efektu polega na wylosowaniu kolorowych prostokątów w niedalekiej odległości od wskazanego punktu. Wektor ruchu poszczególnych elementów tworzony jest poprzez wektor między wylosowanym punktem, a wskazanym jako początkowy.

W metodzie `update(...)` aktualizujemy pozycję prostokątów oraz aktualizujemy wektor, aby wyglądało jakby działa siła ciążenia.

4.1.3.2 Kształty



Rysunek 4.5 Diagram przedstawiający elementy pakietu eu.catengine.renderable.shape.

Pakiet `eu.catengine.renderable.shape` zawiera podstawowe elementy rozszerzające klasę `Shape`.

4.1.3.2.1 Shape

Klasa `Shape` służy za wysokopoziomowy prototyp do tworzenia obiektów sceny. Zawiera domyślne metody do zmiany pozycji, rotacji, koloru oraz skali obiektów. Dodatkowo istnieje możliwość przypisania obiektów nasłuchujących tzw. listenerów. Dziedziczy po klasie `Renderable`.

Przykłady użycia

Tworzenie obiektów odbywa się analogicznie do klasy `Renderable`. Z tą różnicą, że klasa `Shape` implementuje dużo więcej domyślnych metod dla danego obiektu.

4.1.3.2.2 Line

Klasa `Line` przechowuje dane oraz metody linii wyświetlanej w scenie. Oprócz metod podstawowych udostępnia możliwość zmiany grubości rysowanej linii. Dziedziczy po klasie `Shape`.

Przykłady użycia

- Tworzenie linii w scenie.

```
1    @Override
2    public void onPointerMove(TouchPointer pointer, TouchPointer
         historicalPointer)
3    {
4       getSceneTree().addObject(new Line(pointer.getSceneTouchPoint(),
            historicalPointer.getSceneTouchPoint()));
5    }
```

Kod źródłowy 4.8 Przykład użycia klasy Line.

Powyższy kod tworzy linię pomiędzy punktami po których porusza się wskaźnik na ekranie. Zmienna `historicalPointer` określa ostatni punkt w jaki znajdował się wskaźnik, natomiast `pointer` zawiera jego aktualne położenie.

4.1.3.2.3 Rectangle

Klasa `Rectangle` przechowuje dane oraz metody prostokąta wyświetlanego w scenie. Dziedziczy po klasie `Shape`.

Przykłady użycia

- Tworzenie prostokąta oraz jego rotacja uzależniona względem czasu.

```
1  Rectangle rectangle;
2
3  public DemoScene(String sceneName)
4  {
5     super(sceneName);
```

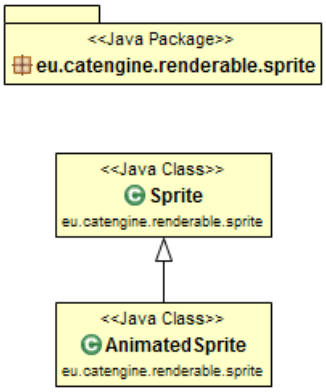
```
6       rectangle = new Rectangle(500, 500, 100, 150);
7       getSceneTree().addObject(rectangle);
8   }
9
10  @Override
11  public void updateScene(float deltaTime, long sceneTime)
12  {
13      rectangle.rotateByAtCenterPoint(deltaTime / 20.0f);
14  }
```

Kod źródłowy 4.9 Przykład użycia klasy Rectangle.

W konstruktorze sceny `DemoScene` stworzony został prostokąt w punkcie (500, 500) o szerokości 100 oraz wysokości 150 pikseli. Następnie został on dodany do drzewa sceny.

W metodzie updateScene(...) zmienia się rotacja obiektu o pewną wartość uzależnioną od czasu trwania pojedynczej klatki. Dzięki temu istnieje możliwość dokładnej kontroli animacji względem czasu sceny.

4.1.4 Elementy graficzne



Rysunek 4.6 Diagram przedstawiający elementy pakietu eu.catengine.renderable.sprite.

Klasy zawarte w pakiecie `eu.catengine.renderable.sprite` w wyniku swoich działań wyświetlają grafikę opartą na teksturach. Pakiet dzieli się na dodatkowe pakiety takie jak kształty, elementy graficzne oraz tekstowe.

4.1.4.1 Sprite

Klasa `Sprite` przechowuje dane oraz metody elementu sprite wyświetlanego w scenie. Dziedziczy po klasie `Rectangle`.

Przykłady użycia

- Tworzenie sprite'a oraz zmiana tekstury po dotknięciu obiektu wskaźnikiem.

```
1  Sprite sprite;
2
3  public DemoScene(String sceneName)
4  {
5     super(sceneName);
6     sprite = new Sprite(500, 500, getTextureManager().get("demo-texture"));
7  }
8
9  @Override
10 public void onPointerDown(TouchPointer pointer)
11 {
12    if(sprite.isAtPoint(pointer.getSceneTouchPoint()))
13    {
14        sprite.setTexture(getTextureManager().get("demo-texture-2"));
15    }
16 }
```

Kod źródłowy 4.10 Przykład użycia klasy Sprite.

W konstruktorze sceny tworzony jest sprite w punkcie (500, 500) o teksturze *demo-texture*. W tym przypadku rozmiar sprite'a pobierany jest automatycznie z rozmiaru tekstury.

Metoda `onPointerDown(...)` sprawdza czy obiekt występuje w punkcie przyłożenia wskaźnika do ekranu, jeśli tak to zmieniamy teksturę obiektu na inną.

4.1.4.2 AnimatedSprite

Klasa `AnimatedSprite` przechowuje dane oraz metody animacji wyświetlanej w scenie. Posiada mechanizmy do kontroli oraz automatycznego usuwania animacji ze sceny po jej zakończeniu. Dziedziczy po klasie `Sprite`.

Ważne elementy

- Aktualizacja oraz wyświetlanie animacji.

```
1  @Override
2  public void update(long elapsedTime)
3  {
4     if (!mVisible)
5     {
6        return;
7     }
8
9     mActualLoop = (elapsedTime - mStartTime) / mTextureAnimation.getDuration();
10    mActualFrame = ((elapsedTime - mStartTime) / mTextureAnimation.
          getFrameDuration()) % mTextureAnimation.getFrames();
```

```
11      mTextureRegion = mTextureAnimation.getRegion(mActualFrame);
12      mNotCreatedInScene = mActualFrame < 0 ? true : false;
13
14      if(mNotCreatedInScene)
15      {
16          return;
17      }
18      else if (mActualFrame == 0 && mActualLoop == 0 && mAnimationListener != null)
19      {
20          mAnimationListener.onAnimationStarted(this);
21      }
22      else if (mActualFrame == 0 && mActualLoop != 0 && mAnimationListener != null)
23      {
24          mAnimationListener.onNewAnimationCycle(this, mActualLoop);
25      }
26
27      boolean keepInScene = (mActualLoop < mMaxLoops || mMaxLoops == -1);
28
29      if (mKeepInScene != keepInScene)
30      {
31          mKeepInScene = keepInScene;
32
33          if (mAnimationListener != null && mKeepInScene == false)
34          {
35            mAnimationListener.onAnimationEnded(this);
36            return;
37          }
38      }
39  }
40
41  @Override
42  public void draw(GL10 gl)
43  {
44      if(mNotCreatedInScene)
45      {
46         return;
47      }
48
49      super.draw(gl);
50  }
```

Kod źródłowy 4.11 Aktualizacja animacji w klasie AnimatedSprite.

Aktualizacja animacji sprowadza się do wyznaczenia jej aktualnej klatki. Wyznaczana jest ona względem czasu jaki minął w scenie od momentu dodania do niej obiektu. Ilość wykonanych pętli obliczana jest analogicznie do aktualnej klatki, z tą różnicą, że wartość ta jest dzielona przez długość całej animacji określonej w milisekundach.

Aktualna klatka jest wyznaczana dzięki wyznaczeniu reszty z dzielenia aktualnej pętli przez ilość klatek animacji. Biorąc pod uwagę to, iż w scenie można kontrolować czas należy sprawdzić czy aktualna klatka nie ma wartości ujemnej. Jeśli tak by się stało, animacja działaby nieprawidłowo. W tym celu dodana została dodatkowa zmienna, dzięki której wiadomo, czy w danym momencie sceny należy wyświetlić animację.

Tworząc obiekt animacji określana jest jej maksymalna ilość pętli. Liczba -1 określa, iż ma się ona wykonywać przez cały czas działania gry. W linii numer 27 utworzona została zmienna `keepInScene`. W przypadku, gdy animacja ma nadal się wykonywać

będzie miała ona wartość `true`, w przeciwnym razie `false`. W następnej instrukcji sprawdzamy czy aktualna wartość zmiennej lokalnej jest różna od wartości zmiennej w klasie. Tylko w przypadku różnicy nadpisujemy wartość zmiennej `mKeepInScene`.

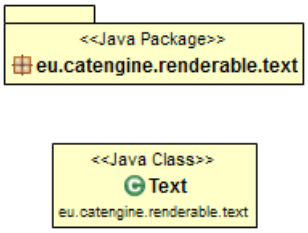
Przykłady użycia

- Tworzenie animacji w scenie oraz dodawanie obiektu nasłuchującego.

```
1  AnimatedSprite animatedSprite;
2  AnimationListener animationListener;
3
4  public DemoScene(String sceneName)
5  {
6     super(sceneName);
7
8     animatedSprite = (AnimatedSprite) getSceneTree().addObject(new AnimatedSprite
           (100, 100, getAnimationManager().get("demo-animation"), -1));
9     animatedSprite.setAnimationListener(new AnimationListener()
10    {
11        @Override
12        public void onAnimationStarted(AnimatedSprite animatedSprite)
13        {
14          Log.d("Debug", "Animation␣started");
15        }
16
17        @Override
18        public void onNewAnimationCycle(AnimatedSprite animatedSprite, long
              lastCycleNumber)
19        {
20          Log.d("Debug", "Actual␣animation␣cycle:␣" + (lastCycleNumber + 1));
21        }
22
23        @Override
24        public void onAnimationEnded(AnimatedSprite animatedSprite)
25        {
26          // TODO Auto-generated method stub
27        }
28    });
29 }
```

Kod źródłowy 4.12 Przykład użycia klasy AnimatedSprite.

W konstruktorze sceny tworzymy oraz jednocześnie dodajemy do sceny animację w punkcie (100, 100) korzystającą z gotowego wzoru pobranego z `AnimationManager` o nazwie *demo-animation*. Określamy jedynie ilość pętli jaką ma ona wykonać, w tym przypadku ma się ona wykonywać non stop. Następnie dodajemy obiekt nasłuchujący animację. Zawarte zostały w nim instrukcje, które wyświetlają kiedy animacja została uruchomiona oraz jaka jest jej aktualnie wykonywana pętla.



Rysunek 4.7 Diagram przedstawiający elementy pakietu eu.catengine.renderable.text.

4.1.5 Elementy tekstowe

Elementy pakietu `eu.catengine.renderable.text` służą do wyświetlania elementów tekstowych w scenie.

4.1.5.1 Text

Klasa `Text` przechowuje dane oraz metody dynamicznego tekstu wyświetlanego w scenie. Dziedziczy po klasie `Shape`.

Ważne elementy

- Rysowanie tekstu w scenie z uwzględnieniem nowych linii.

```
1  @Override
2  public void draw(GL10 gl)
3  {
4     if (mVisible == false)
5         return;
6
7     mNewLines = 0;
8     this.moveTo(mOriginPoint);
9
10    gl.glPushMatrix();
11    gl.glEnable(GL10.GL_TEXTURE_2D);
12    gl.glEnable(GL10.GL_BLEND);
13    gl.glEnable(GL10.GL_ALPHA_BITS);
14    gl.glBlendFunc(GL10.GL_SRC_ALPHA, GL10.GL_ONE_MINUS_SRC_ALPHA);
15    gl.glColor4f(mColor.red, mColor.green, mColor.blue, mColor.alpha);
16    gl.glBindTexture(GL10.GL_TEXTURE_2D, mFont.getTexturePointer());
17    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
18    gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
19    gl.glFrontFace(GL10.GL_CW);
20
21    for (int i = 0; i < mText.length(); i++)
22    {
23        mCurrentChar = mText.charAt(i);
24
25        if (mCurrentChar == '\n')
26        {
27          mNewLines++;
```

```
28          this.moveTo(new Point(mOriginPoint).add(0, -mNewLines * mFont.
                getCharHeight() * getScaleY()));
29          continue;
30        }
31
32        mCurrentString = mText.substring(i, i + 1);
33        mTextureRegion = mFont.getTextureRegion(mCurrentString);
34
35        if (mTextureRegion == null)
36        {
37          mCurrentChar = '#';
38          mTextureRegion = mFont.getTextureRegion("#");
39        }
40
41        gl.glVertexPointer(2, GL10.GL_FLOAT, 0, mVertexBuffer);
42        gl.glTexCoordPointer(2, GL10.GL_FLOAT, 0, mTextureRegion.getTextureBuffer
              ());
43        gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, mVerticesCount);
44
45        this.moveBy(new Vector(mFont.getCharWidth(mCurrentChar) * getScaleX(), 0))
              ;
46    }
47
48    gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
49    gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
50    gl.glDisable(GL10.GL_ALPHA_BITS);
51    gl.glDisable(GL10.GL_BLEND);
52    gl.glDisable(GL10.GL_TEXTURE_2D);
53    gl.glPopMatrix();
54  }
```

Kod źródłowy 4.13 Metoda rysowanie tekstu w klasie Text.

Renderowanie pojedynczych znaków na ekranie odbywa się dzięki klasie `Font`, która zostanie opisana bardziej szczegółowo w dalszej części pracy. Klasa `Text` odpowiada za miejsce, w których każdy ze znaków jest wyświetlany.

Metoda `draw(...)` swoje działanie rozpoczyna od ustawienia wartości odpowiadającej za ilość nowych linii na zero oraz określenia pozycji początkowej kursora. Następnie wywołuje metody konfigurujące środowisko OpenGL dla danego obiektu, po których następuje pętla `for`. Dla każdego elementu zmiennej `mText` sprawdzamy, czy jest on znakiem nowej linii, jeśli tak to przesuwamy kursor w dół o wysokość danej czcionki. Jeśli nie, pobieramy region tekstury odpowiedzialny za dany znak. W przypadku, gdy taki znak nie posiada swojego regionu zamieniany jest on na `#`. Przed końcem pętli dany element jest renderowany w scenie oraz kursor zostaje przesunięty o jego szerokość. Ostatecznie wywołujemy metody środowiska OpenGL kończące pracę z danym obiektem.

Przykłady użycia

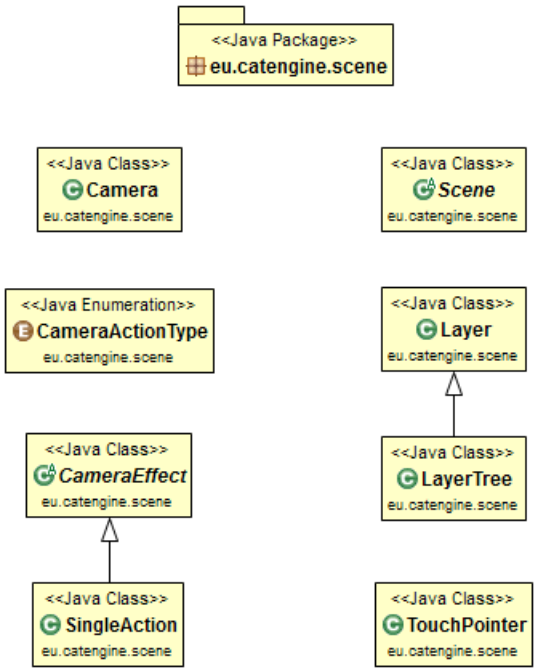
- Wyświetlanie ilości wyświetlanych klatek na sekundę.

```
1  Text actualFPS;
2
3  public DemoScene(String sceneName)
4  {
5     super(sceneName);
6
7     actualFPS = (Text) getSceneTree().addObject(new Text("Please␣wait...", 0,
           754, getFontManager().get("Verdana", 12)));
8     FPSCounter.startCounter();
9  }
10
11 @Override
12 public void updateScene(float deltaTime, long sceneTime)
13 {
14    FPSCounter.update();
15    actualFPS.setText(FPSCounter.getFPS() + "");
16 }
```

Kod źródłowy 4.14 Przykład użycia klasy Text.

W konstruktorze sceny tworzymy tekst w punkcie (0, 754) o czcionce pod nazwą `Verdana` o wielkości 12 punktów. Następnie uruchamiamy licznik klatek. W metodzie `updateScene(...)` aktualizujemy obiekt licznika oraz tekst, który chcemy wyświetlić na ekranie.

4.1.6 Scena oraz jej elementy



Rysunek 4.8 Diagram przedstawiający elementy pakietu eu.catengine.scene.

Pakiet `eu.catengine.scene` zawiera w sobie klasę sceny oraz inne elementy, które są z nią ściśle powiązane.

4.1.6.1 Scene

Klasa `Scene` jest jedną z najbardziej rozbudowanych klas w przedstawianym silniku. Odpowiada za wyświetlanie wszystkich obiektów na ekranie urządzenia. Tworząc nową aplikację należy utworzyć własny obiekt, który będzie dziedziczyć elementy po klasie `Scene`. Ostatecznie należy dodać go do `SceneManager`. W ten sposób tworzone są poszczególne elementy gry.

Ważne elementy

- Pętla gry z uwzględnieniem kontroli czasu sceny.

```
1  public void updateRunningScene()
2  {
3     if (!mSceneIsRunning)
4     {
5         return;
6     }
7
8     mLastFrameDuration = System.currentTimeMillis() - mSceneStartTime -
          mLastUpdateTime;
9     mLastUpdateTime += mLastFrameDuration;
10    mAccumulator += mLastFrameDuration - mScenePauseDuration;
11    mAccumulator = Math.min(mMaxTimeAccumulator, Math.max(0, mAccumulator));
12    mFrameDurationIncSpeed = mFrameDuration / Math.abs(mSceneSpeed);
13    mFrameDurationDelta = mFrameDuration * Math.signum(mSceneSpeed);
14    mScenePauseDuration = 0;
15
16    while (mAccumulator >= mFrameDurationIncSpeed)
17    {
18        mElapsedTime += mFrameDurationDelta;
19
20        mCamera.update();
21        mInterpolator.update();
22        mBackgroundTree.updateObjects(mElapsedTime);
23        mHudTree.updateObjects(mElapsedTime);
24        mSceneTree.updateObjects(mElapsedTime);
25
26        updateScene(mFrameDurationDelta, mElapsedTime);
27
28        mAccumulator -= mFrameDurationIncSpeed;
29    }
30 }
```

Kod źródłowy 4.15 Implementacja pętli gry zawartej w Scenie.

Opis powyższego kod należy rozpocząć od wyjaśnienia znaczenia zmiennych zawartych w liniach od 8 do 13.

- `mSceneStartTime` - czas dodania sceny do managera,
- `mLastFrameDuration` - czas trwania ostatniej klatki,
- `mLastUpdateTime` - czas ostatniej aktualizacji sceny,

- `mScenePauseDuration` - czas trwania sceny w stanie pauzy,
- `mMaxTimeAccumulator` - maksymalny czas o jaki scena może zostać zaktualizowana,
- `mAccumulator` - czas jaki minął od ostatniej aktualizacji z uwzględnieniem pauzy,
- `mFrameDurationIncSpeed` - czas trwania klatki z uwzględnieniem zmiany prędkości czasu,
- `mFrameDurationDelta` - różnica czasu między ostatnią, a obecną klatką.

Aktualizacja elementów sceny odbywa się dzięki pętli `while`. Wykonuje ona instrukcje w niej zawarte tylko i wyłącznie jeśli czas jaki minął od ostatniej aktualizacji jest większy bądź równy od czasu trwania pojedynczej klatki z uwzględnieniem współczynnika przyspieszenia sceny. Mówiąc krócej scena jest aktualizowana w momencie, gdy czas od ostatniej aktualizacji pozwala na wyświetlenie następnej klatki.

Kontrola czasu w scenie odbywa się poprzez zmienną określającą współczynnik przyspieszenia. Dla współczynnika równego czas upływa normalnie, dla 2.0 upływałby dwa razy szybciej, dla 0.5 dwa razy wolniej. Należy również wspomnieć o tym, iż niezależnie od wartości współczynnika przyspieszenia wszystkie wywołania metody `updateScene(...)` będą odbywały się metodą stałokrokową.

Przykłady użycia

- Tworzenie prostej gry przy użyciu sceny.

```
1  public class DemoScene extends Scene
2  {
3      Random random = new Random();
4      Font font = getFontManager().get("Verdana", 35);
5      ScreenManager screenManager = getScreenManager();
6
7      Camera camera = getCamera();
8      LayerTree root = getSceneTree();
9      LayerTree hud = getHudTree();
10
11     Rectangle player1, player2, ball;
12     Rectangle[] borders;
13     Text player1Text, player2Text;
14     Vector ballVector;
15
16     int player1Score = 0, player2Score = 0;
17
18     float playerMaxSpeed = 10.0f;
19     float screenWidth = screenManager.getScreenWidth();
20     float screenHeight = screenManager.getScreenHeight();
21
22     boolean player1Down, player1Up, player2Down, player2Up;
23
```

```
24      public DemoScene(String sceneName)
25      {
26        super(sceneName);
27
28        player1 = new Rectangle(40, screenHeight / 2, 20, 150);
29        player2 = new Rectangle(screenWidth - 60, screenHeight / 2, 20, 150);
30        ball = new Rectangle(screenWidth / 2 - 10, screenHeight / 2 + 10, 20, 20);
31
32        borders = new Rectangle[5];
33        borders[0] = new Rectangle(0, screenHeight, 10, screenHeight);
34        borders[1] = new Rectangle(screenWidth - 10, screenHeight, 10, screenHeight
              );
35        borders[2] = new Rectangle(0, 10, screenWidth, 10);
36        borders[3] = new Rectangle(0, screenHeight - 100, screenWidth, 10);
37        borders[4] = new Rectangle(0, screenHeight, screenWidth, 10);
38
39        player1Text = new Text("0", 20, screenHeight - 32, font);
40        player2Text = new Text("0", screenWidth - 20 - font.getTextWidth("0"),
              screenHeight - 32, font);
41
42        ballVector = new Vector((float) (5 * Math.signum(random.nextFloat() - 0.5))
              , (float) (5 * Math.signum(random.nextFloat() - 0.5)));
43
44        root.addObject(player1);
45        root.addObject(player2);
46        root.addObject(ball);
47
48        for (int i = 0; i < 5; i++)
49        {
50           root.addObject(borders[i]);
51        }
52
53        hud.addObject(player1Text);
54        hud.addObject(player2Text);
55        hud.addObject(new Text("SCORE", (screenWidth - font.getTextWidth("SCORE"))
              / 2, screenHeight - 32, font));
56
57        camera.moveTo(0, screenHeight);
58      }
59
60      private void updateInputs(TouchPointer pointer)
61      {
62        Point touchPoint = pointer.getSceneTouchPoint();
63
64        if (touchPoint.x < screenWidth / 2)
65        {
66           if (touchPoint.y < screenHeight / 2)
67           {
68              player1Down = true;
69              player1Up = false;
70           }
71           else
72           {
73              player1Up = true;
74              player1Down = false;
75           }
76        }
77        else
78        {
79           if (touchPoint.y < screenHeight / 2)
80           {
81              player2Down = true;
82              player2Up = false;
83           }
84           else
85           {
86              player2Up = true;
```

```
87              player2Down = false;
88            }
89          }
90        }
91
92        @Override
93        public void onPointerDown(TouchPointer pointer)
94        {
95          updateInputs(pointer);
96        }
97
98        @Override
99        public void onPointerMove(TouchPointer pointer, TouchPointer
              historicalPointer)
100       {
101         updateInputs(pointer);
102       }
103
104       @Override
105       public void onPointerUp(TouchPointer pointer)
106       {
107         Point touchPoint = pointer.getSceneTouchPoint();
108
109         if (touchPoint.x < screenWidth / 2)
110         {
111           player1Down = player1Up = false;
112         }
113         else
114         {
115           player2Down = player2Up = false;
116         }
117       }
118
119       @Override
120       public void onChangeToInactiveScene(){}
121
122       @Override
123       public void onChangeToActiveScene(){}
124
125       @Override
126       public Boolean onButtonDown(int keyCode, KeyEvent event){ return false; }
127
128       @Override
129       public void updateScene(float deltaTime, long sceneTime)
130       {
131         if (player1Up && (player1.getAnchorPoint().y + playerMaxSpeed) <
                screenHeight - 110)
132         {
133           player1.moveBy(0, playerMaxSpeed);
134         }
135         else if (player1Down && (player1.getAnchorPoint().y - player1.getHeight() -
                playerMaxSpeed) > 10)
136         {
137           player1.moveBy(0, -playerMaxSpeed);
138         }
139
140         if (player2Up && (player2.getAnchorPoint().y + playerMaxSpeed) <
                screenHeight - 110)
141         {
142           player2.moveBy(0, playerMaxSpeed);
143         }
144         else if (player2Down && (player2.getAnchorPoint().y - player2.getHeight() -
                playerMaxSpeed) > 10)
145         {
146           player2.moveBy(0, -playerMaxSpeed);
147         }
148
```

```
149         // Walls (top and bottom)
150         if (ball.getAnchorPoint().y - ball.getHeight() + ballVector.y < 10)
151         {
152            ballVector.y *= -1;
153         }
154         else if (ball.getAnchorPoint().y + ballVector.y > screenHeight - 110)
155         {
156            ballVector.y *= -1;
157         }
158
159         // Players
160         if (player1.isAtPoint(ball.getAnchorPoint()))
161         {
162            ballVector.x *= -1;
163         }
164         else if (player2.isAtPoint(new Point(ball.getAnchorPoint()).add(20, 0)))
165         {
166            ballVector.x *= -1;
167         }
168
169         // Points
170         if (ball.getAnchorPoint().x < 10)
171         {
172            player2Score++;
173            player2Text.setText(player2Score + "");
174            player2Text.moveTo(screenWidth - 20 - font.getTextWidth(player2Score + "
                  "), screenHeight - 32);
175
176            ball.moveTo(screenWidth / 2 - 10, screenHeight / 2 + 10);
177            ballVector = new Vector((float) (5 * Math.signum(random.nextFloat() -
                  0.5)), (float) (5 * Math.signum(random.nextFloat() - 0.5)));
178         }
179         else if (ball.getAnchorPoint().x + ball.getWidth() > screenWidth - 10)
180         {
181            player1Score++;
182            player1Text.setText(player1Score + "");
183
184            ball.moveTo(screenWidth / 2 - 10, screenHeight / 2 + 10);
185            ballVector = new Vector((float) (5 * Math.signum(random.nextFloat() -
                  0.5)), (float) (5 * Math.signum(random.nextFloat() - 0.5)));
186         }
187
188         ball.moveBy(ballVector);
189
190         // Speed up
191         setSceneSpeed(1.0f + (sceneTime / 100000.0f));
192      }
193   }
```

Kod źródłowy 4.16 Przykład użycia klasy Scene.

Przedstawiona klasa `DemoScene` dziedziczy po klasie `Scene`. W tym przypadku posłużyła ona za stworzenie prostej gry o nazwie *Pong*.

Zasady gry są proste. Posiada ona dwóch graczy oraz piłkę, która porusza się między graczami. Należy odbijać piłeczkę do momentu, aż któryś z użytkowników jej nie odbije. Wówczas naliczany zostaje punkt dla przeciwnika.

W liniach od 3 do 22 występują zmienne pomocnicze, które znacznie przyspieszają pracę oraz upraszczają kod. Konstruktor klasy tworzy wszystkie obiekty sceny. Plansza

gry została zaprojektowana tak, że jeden z graczy znajduje się po lewej, drugi po prawej stronie. Dookoła całego widoku kamery utworzona została ramka ograniczająca pole, w którym może poruszać się piłka. W górnej części znajdują się teksty informacyjne wyświetlające aktualny wynik. Sposób poruszania się piłeczki jest dobierany w sposób pseudolosowy.

Sterowanie graczy w metodach `onPointerDown(...)` oraz `onPointerMove(...)` sprowadza się do wywołania `updateInputs(...)`. Metoda ta sprawdza, w której części ekranu został wykryty punkt dotyku. Jeśli w lewej części ekranu to dotyczy on gracza pierwszego (znajdującego się po lewej stronie planszy) i odwrotnie dla gracza drugiego. Następnie sprawdzane jest czy wspominany punkt znajduje się w górnej czy dolnej części ekranu. W zależności od tego gracz porusza się w odpowiednim kierunku.

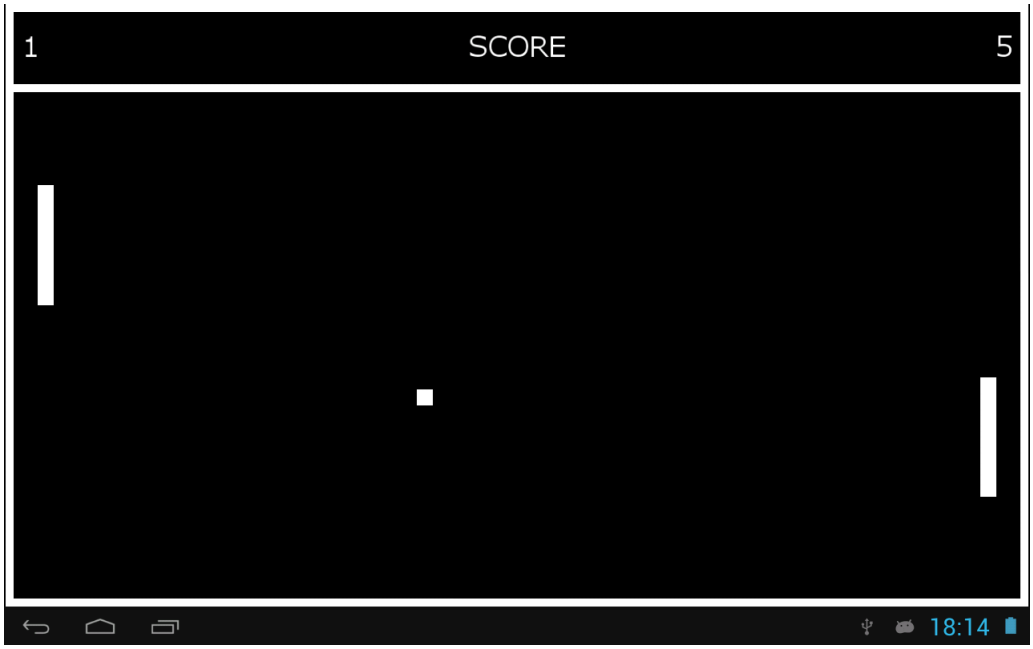
Metody `onChangeToInactiveScene()` oraz `onChangeToActiveScene()` w tym przypadku nie zawierają żadnych instrukcji. Wywoływane są one w momencie, gdy obecna scena jest zmieniana na nieaktywną lub na aktywną. Można tutaj odświeżać widok lub na nowo ustawiać pewne elementy sceny.

Metoda `onButtonDown(...)` obsługuje akcje dotyczące fizycznych przycisków na urządzeniu. W tym przypadku zwracamy wartośc `false` informując system Android, iż nie obsłużyliśmy danej akcji. Wywołana zostanie wówczas domyślna akcja dla aplikacji powiązana z wybranym przyciskiem.

Aktualizacja elementów sceny w metodzie `updateScene(...)` ogranicza się do zmiany oraz sprawdzenia pozycji graczy oraz piłki. Należy sprawdzać czy nie występują zderzenia między piłką i ścianą lub graczem. W takich przypadkach zmieniamy wektor ruchu na odwrotny dla poszczególnej składowej, której dotyczy zderzenie. W przypadku, gdy piłka wyjdzie poza obszar danego gracza naliczane są punkty dla przeciwnika. Ostatecznie dla urozmaicenia przyspieszamy czas gry o pewną wartość uzyskaną na podstawie aktualnego czasu sceny.

4.1.6.2 Layer, LayerTree

Klasa `Layer` przechowuje wszystkie elementy sceny, które mają zostać wyświetlone na ekranie. Ma ona możliwość tworzenia warstw dla obiektów. Dodatkowo każda warstwa może zawierać dowolną liczbę obiektów oraz warstw potomnych. Jedyną zasadą jest to, iż dany obiekt może wystąpić w warstwie oraz jej warstwach potomnych tylko raz. Oprócz tego klasa



Rysunek 4.9 Scena implementująca grę *Pong*.

ta sortuje elementy względem momentu dodania do sceny w taki sposób, aby ostatnio dodany obiekt znajdował się na wierzchu. Posiada również metody, dzięki którym w łatwy sposób można uzyskać dowolną warstwę czy obiekt sceny poprzez jej nazwę.

`LayerTree` rozszerza klasę `Layer` o dodatkowe zmienne oraz metody, dzięki którym scena renderuje obiekty. Służy ona za pojedynczy wierzchołek drzewa. W każdej scenie występują dokładnie trzy takie elementy dla obiektów:

- tła - uzyskiwane dzięki metodzie `getBackgroundTree()`,
- sceny - uzyskiwane dzięki metodzie `getSceneTree()`,
- hud (ang. head up display) - uzyskiwane dzięki metodzie `getHudTree()`.

Przykłady użycia

- Zarządzanie obiektami sceny poprzez warstwy.

```
1    Layer bottomLayer = getSceneTree().addLayer("warstwa-1");
2    Layer topLayer = getSceneTree().addLayer("warstwa-2");
3
4    bottomLayer.addLayer("warstwa-1-1");
5    bottomLayer.addLayer("warstwa-1-2");
6
7    topLayer.addLayer("warstwa-2-1");
8
```

```
9     bottomLayer.getLayer("warstwa-1-1").addObject(new Rectangle("rec1", 10, 10,
          100, 100));
10    bottomLayer.getLayer("warstwa-1-2").addObject(new Rectangle("rec2", 100, 100,
           100, 100));
11
12    Layer warstwaObiektuRec1 = getSceneTree().getObjectLayer("rec1");
13
14    warstwaObiektuRec1.deleteLayer();
15    getSceneTree().deleteLayer("warstwa-1");
16    getSceneTree().deleteLayer("warstwa-2");
```

Kod źródłowy 4.17 Przykład użycia klasy Layer oraz TreeLayer.

Powyższy kod przedstawia podstawowe operacje na warstwach. Tworzenie oraz dodawanie nowych warstw oraz obiektów znajduje się w liniach od 1 do 7. Zaraz po tym występuje metoda, dzięki której można odnaleźć warstwę, do której przynależy wybrany obiekt. Ostatnie trzy linie usuwają wcześniej stworzone warstwy.

Istnieją dwa sposoby usuwania warstw. Pierwszy z nich, zakłada usuwanie samego siebie, stąd `warstwaObiektuRec1.deleteLayer()`. Druga metoda służy do usuwania warstwy bezpośrednio z wierzchołka drzewa wskazując jej nazwę poprzez parametr.

4.1.6.3 Camera

Klasa `Camera` przechowuje dane oraz metody dotyczące kamery w scenie. Należy pamiętać o tym, że dla każdej sceny istnieje jeden obiekt kamery. Oprócz tego Camera posiada wewnętrzne klasy, dzięki którym można rozbudować sposób poruszania kamerą o dodatkowe efekty. Daje nam również możliwość zablokowania ruchów kamery do wyznaczonego obszaru.

Przykłady użycia

- Poruszanie kamerą wewnątrz sceny.

```
1  @Override
2  public void onPointerMove(TouchPointer pointer, TouchPointer historicalPointer)
3  {
4     Point p1 = historicalPointer.getFixedTouchPoint();
5     Point p2 = pointer.getFixedTouchPoint();
6
7     getCamera().moveBy(p2.x - p1.x, p1.y - p2.y, 100);
8  }
```

Kod źródłowy 4.18 Przykład użycia klasy Camera.

Powyższy kod przedstawia jeden ze sposobów w jaki można poruszać kamerą w scenie. W tym przypadku pobierane są dwa punkty, dzięki którym tworzony jest wektor

ruchu kamery. Dla upłynnienia tego efektu użyta została funkcja `moveBy(...)`, która przesuwa widok kamery o dany wektor w czasie podanym jako parametr.

4.1.6.4 TouchPointer

Klasa `TouchPointer` przechowuje dane dotyczące wskaźnika, który wywołuje pewną interakcję z ekranem dotykowym. Udostępnia jedynie cztery metody:

- `int getPointerId()` - zwraca id wskaźnika,
- `Point getRawTouchPoint()` - zwraca punkt przechwycony bezpośrednio z funkcji systemu Android,
- `Point getSceneTouchPoint()` - Zwraca punkt dostosowany do współrzędnych sceny,
- `Point getFixedTouchPoint()` - zwraca punkt dostosowany do elementów, które nie zmieniają pozycji na ekranie, takich jak tło lub elementy HUD.

Przykłady użycia

- Rysowanie linii z uwzględnieniem dotyku wielopunktowego.

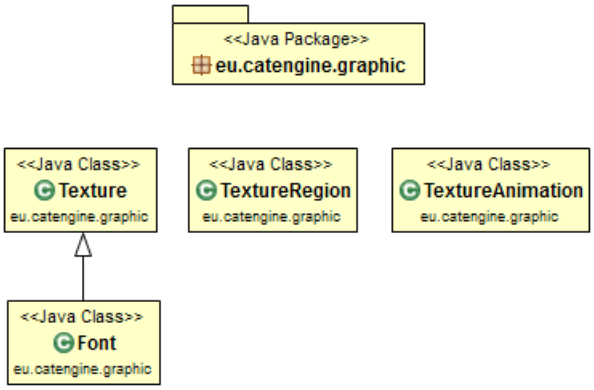
```
1 @Override
2 public void onPointerMove(TouchPointer pointer, TouchPointer historicalPointer)
3 {
4    Point p1 = historicalPointer.getSceneTouchPoint();
5    Point p2 = pointer.getSceneTouchPoint();
6
7    getSceneTree().addObject(new Line(p1, p2));
8 }
```

Kod źródłowy 4.19 Przykład użycia klasy TouchPointer.

Powyższy kod pobiera dwa kolejne punkty, które znajdują się na drodze wskaźnika poruszanego po ekranie. Następnie do sceny dodaje linię zbudowaną między wspomnianymi punktami.

4.1.7 Obsługa tekstur

Pakiet `eu.catengine.graphic` zawiera elementy odpowiedzialne za obsługę tekstur dla obiektów sceny.



Rysunek 4.10 Diagram przedstawiający elementy pakietu eu.catengine.graphic.

4.1.7.1 Texture

Klasa `Texture` odpowiedzialna jest za mapowanie oraz bindowanie tekstur w środowisku OpenGL. Oprócz tego posiada metody, dzięki którym można odczytywać dowolne informacje związane z obiektem oraz tworzyć dodatkowe obszary wewnątrz tekstur, zwane regionami. Należy pamiętać o tym, aby obiekty tworzyć poprzez `TextureManager` na samym początku pracy silnika. Dzięki temu unika się wielokrotnego ładowania dużych plików graficznych.

Ważne elementy

- Wczytywanie oraz dostosowywanie tekstur dla biblioteki OpenGL.

```
1  public Texture(GL10 gl, Bitmap bitmap)
2  {
3     mTextureRegionHashMap = new HashMap<String, TextureRegion>();
4
5     float mRegionCoords[] = new float[] {
6        0.0f, 1.0f,   // top left (V2)
7        0.0f, 0.0f,   // bottom left (V1)
8        1.0f, 1.0f,   // top right (V4)
9        1.0f, 0.0f    // bottom right (V3)
10    };
11
12    mImageSize = new Size(bitmap.getWidth(), bitmap.getHeight());
13
14    int textureWidth = (int) mImageSize.width;
15    int textureHeight = (int) mImageSize.height;
16
17    if (!isPowerOfTwo(textureWidth))
18        textureWidth = nextPowerOfTwo(textureWidth);
19    if (!isPowerOfTwo(textureHeight))
20        textureHeight = nextPowerOfTwo(textureHeight);
21
22    if (textureWidth != bitmap.getWidth() || textureHeight != bitmap.getHeight())
23    {
```

```
24          Bitmap tmpBitmap = Bitmap.createBitmap(textureWidth, textureHeight, Config
                .ARGB_8888);
25          Canvas canvas = new Canvas(tmpBitmap);
26
27          mRegionCoords[1] = (float) bitmap.getHeight() / (float) textureHeight;
28          mRegionCoords[5] = (float) bitmap.getHeight() / (float) textureHeight;
29
30          mRegionCoords[4] = (float) bitmap.getWidth() / (float) textureWidth;
31          mRegionCoords[6] = (float) bitmap.getWidth() / (float) textureWidth;
32
33          canvas.drawBitmap(bitmap, new Matrix(), null);
34          bitmap = tmpBitmap;
35      }
36
37      mTextureSize = new Size(textureWidth, textureHeight);
38
39      gl.glGenTextures(1, mTexture, 0);
40      gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexture[0]);
41      gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, GL10.
            GL_NEAREST);
42      gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, GL10.
            GL_LINEAR);
43
44      GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
45
46      bitmap.recycle();
47
48      mTextureRegionHashMap.put("default", new TextureRegion(mRegionCoords));
49  }
```

Kod źródłowy 4.20 Implementacja wczytywania tekstur.

Wczytując teksturę poprzez `TextureManager` klasa `Texture` jako parametr otrzymuje referencję na obiekt biblioteki graficznej oraz bitmapę, którą należy mapować.

Początkowo tworzona jest `hash mapa`[1], która będzie przechowywać wszystkie regiony tekstur oraz jeden domyślny region. Następnie do zmiennych pomocniczych wczytywany jest rozmiar wczytanej bitmapy. Jeśli któryś z wymiarów jest różny od potęgi liczby 2 wielkość tekstury jest zwiększana tak, aby spełniała ten warunek.

W linii numer 22 sprawdzane jest, czy wielkość tekstury jest taka sama jak wielkość wczytanej bitmapy. Jeśli przynajmniej jeden z wymiarów jest różny, to bitmapa jest przerysowywana na żądany rozmiar. Tego typu operacja nie zmienia jakości grafiki ze względu na to, iż przerysowywanie odbywa się bez rozciągania obrazu do nowego wymiaru. Proporcje są zachowane dzięki przeliczeniu współrzędnych regionu.

Następnie przypisujemy rozmiar tekstury do zmiennej wewnątrz klasy oraz bindujemy teksturę w pamięci. Po tych operacjach czyścimy pamięć po bitmapie oraz dodajemy domyślny region tekstury do hash mapy wspomnianej na początku.

[1]Hash mapa - inaczej tablica mieszająca. Struktura danych, która jest jednym ze sposobów realizacji tablicy asocjacyjnej, w której zaimplementowano bardzo szybki dostęp do danych.

Przykłady użycia

- Tworzenie obiektów wykorzystujących tekstury.

```
1  @Override
2  protected void loadResources()
3  {
4     getTextureManager().loadTexture("demo-texture.jpg");
5  }
6
7  public DemoScene(String sceneName)
8  {
9     // ...
10
11    Texture texture = getTextureManager().get("demo-texture");
12    getSceneTree().addObject(new Sprite(100, 100, texture));
13 }
```

Kod źródłowy 4.21 Przykład użycia klasy Texture.

Powyższy fragment kodu prezentuje w jaki sposób należy wczytywać obiekty klasy `Texture`. Metoda `loadResources()` pochodzi z klasy rozszerzającej `CatEngineCore` i wczytuje teksturę z pliku `demo-texture.jpg` do managera.

Konstruktor klasy `DemoScene` pobiera wczytaną teksturę z managera, a następnie tworzy obiekt klasy `Sprite`, który wyświetlać będzie wczytaną teksturę w scenie.

4.1.7.2 TextureRegion

Klasa `TextureRegion` służy do przechowania danych dotyczących pojedynczych regionów dowolnej tekstury. Dzięki tego typu rozwiązaniom nie trzeba wczytywać dużej ilości pojedynczych plików jako tekstur. Wystarczy bowiem utworzyć tak zwany atlas tekstur i następnie wskazać pozycje każdego elementu, który w sobie zawiera.

Przykłady użycia

- Tworzenie obiektów wykorzystujących regiony tekstur.

```
1  @Override
2  protected void loadResources()
3  {
4     Texture texture = getTextureManager().loadTexture("demo-texture.jpg");
5     texture.addTextureRegion("demo-region1", new Point(0, 0), new Size(100, 100))
          ;
6     texture.addTextureRegion("demo-region2", new Point(100, 0), new Size(100,
          100));
7  }
8
9  public DemoScene(String sceneName)
```

```
10 {
11    // ...
12
13    Texture texture = getTextureManager().get("demo-texture");
14    getSceneTree().addObject(new Sprite(100, 100, texture, "demo-region2"));
15 }
```

Kod źródłowy 4.22 Przykład użycia klasy TextureRegion.

Powyższy kod prezentuje w jaki sposób należy tworzyć obiekty klasy `TextureRegion`. Metoda `loadResources()` pochodzi z klasy rozszerzającej `CatEngineCore`. Wczytuje teksturę z pliku `demo-texture.jpg` po czym dodawane są dwa regiony.

Konstruktor klasy `DemoScene` pobiera wczytaną teksturę z managera. Następnie wykorzystując drugi z utworzonych regionów tworzy obiekt klasy `Sprite`. Obiekt ten nie będzie wyświetlać całej tekstury, tylko jej konkretny region.

4.1.7.3 TextureAnimation

Klasa `TextureAnimation` przechowuje dane animacji poklatkowych opartych na teksturach oraz regionach z niej wyodrębnionych. Ze względów optymalizacyjnych nie dziedziczy po klasie `Texture` co pozwala zarządzać regionami w zupełnie inny sposób, który znacznie przyspiesza dostęp do danych. Służy za wzorzec do tworzenia obiektów klasy `AnimatedSprite`, w którym określone zostały takie dane jak czas trwania animacji oraz poszczególnej klatki, a także domyślny rozmiar.

Przykłady użycia

- Tworzenie animacji w scenie.

```
1  @Override
2  protected void loadResources()
3  {
4     Texture texture1 = getTextureManager().loadTexture("demo-texture1.jpg");
5     Texture texture2 = getTextureManager().loadTexture("demo-texture2.jpg");
6
7     TextureAnimation textureAnimation1 = getAnimationManager().loadAnimation("
          demo-animation1", texture1, new Size(10, 10), 50, 100);
8     TextureAnimation textureAnimation2 = getAnimationManager().loadAnimation("
          demo-animation2", texture2, new Size(10, 10), 50, 10, 100);
9  }
10
11 public DemoScene(String sceneName)
12 {
13    // ...
14
15    getSceneTree().addObject(new AnimatedSprite("animation1", 0, 0,
          textureAnimation1, -1));
```

```
16      getSceneTree().addObject(new AnimatedSprite("animation2", 0, 0,
            textureAnimation2, 10));
17  }
```

Kod źródłowy 4.23 Przykład użycia klasy TextureAnimation.

Powyższy fragment kodu prezentuje w jaki sposób należy wczytywać obiekty klasy `TextureAnimation`. Metoda `loadResources()` pochodzi z klasy rozszerzającej `CatEngineCore` i wczytuje tekstury, z których tworzone będą wzory do tworzenia animowanych elementów sceny.

Istnieją dwie metody wczytywania prototypów animacji. Pierwsza z nich (zawarta w linii numer 7) zakłada, iż poszczególne klatki animacji ułożone są na teksturze w jednej linii. Druga posiada dodatkowy parametr, który określa ile elementów znajduje się w jednej linii. Dzięki temu jest w stanie wczytać elementy ułożone w prostokąt.

Tworzenie obiektów sceny, które wyświetlać będą odpowiednie animacje zawarte zostały w dwóch ostatnich liniach. Pierwsza animacja o nazwie *animation1* wykonywać się będzie nieustannie w trakcie wyświetlania sceny, druga natomiast wykona się jedynie dziesięć razy po czym zostanie automatycznie usunięta ze sceny.

4.1.7.4 Font

Klasa `Font` służy do przechowywania danych czcionek. W celach optymalizacyjnych każdy obiekt tej klasy reprezentuje czcionkę o wielkości, która została podana managerowi w momencie jej wczytywania. Dziedziczy po klasie `Texture`.

Ważne elementy

- Tworzenie tekstury dla czcionki.

```
1  public Font(GL10 gl, String fontName, Typeface typeFace, float fontSize, String
       dataset)
2  {
3     mFontName = fontName;
4     mTypeFace = typeFace;
5     mFontSize = fontSize;
6
7     mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
8     mPaint.setAntiAlias(true);
9     mPaint.setTypeface(mTypeFace);
10    mPaint.setTextSize(mFontSize);
11    mPaint.setARGB(0xff, 0xff, 0xff, 0xff);
12
13    Paint.FontMetrics fm = mPaint.getFontMetrics();
14    mCharHeight = (float) Math.ceil(Math.abs(fm.bottom) + Math.abs(fm.top));
15    mCharWidth = new HashMap<Character, Float>();
```

```
16      mCharWidthMax = 0;
17
18      char[] s = new char[2];
19      float[] w = new float[2];
20
21      for (int i = 0; i < dataset.length(); i++)
22      {
23          char c = dataset.charAt(i);
24          s[0] = c;
25          mPaint.getTextWidths(s, 0, 1, w);
26
27          mCharWidth.put(c, w[0]);
28
29          if (w[0] > mCharWidthMax)
30          {
31            mCharWidthMax = w[0];
32          }
33      }
34
35      mCellWidth = (int) mCharWidthMax + mPaddingX;
36      mCellHeight = (int) mCharHeight + mPaddingY;
37
38      int field = dataset.length() * mCellWidth * mCellHeight;
39      int tmpWidth = nextPowerOfTwo((int) Math.ceil(Math.sqrt(field)));
40      mTextureSize = new Size(tmpWidth, tmpWidth);
41
42      mItemsInRow = tmpWidth / mCellWidth;
43      mItemsInCol = tmpWidth / mCellHeight;
44
45      if ((mItemsInCol * mItemsInRow) < dataset.length())
46      {
47          mTextureSize.width *= 2;
48          mItemsInRow *= 2;
49      }
50
51      mImageSize = mTextureSize;
52
53      Bitmap bitmap = Bitmap.createBitmap((int) mTextureSize.width, (int)
            mTextureSize.height, Bitmap.Config.ARGB_4444);
54      Canvas canvas = new Canvas(bitmap);
55      bitmap.eraseColor(0);
56
57      for (int i = 0; i < dataset.length(); i++)
58      {
59          int x = (int) ((i % mItemsInRow) * mCellWidth);
60          int y = (int) ((i / mItemsInRow) * mCellHeight) + mCellHeight;
61
62          canvas.drawText(dataset.substring(i, i + 1), x, y, mPaint);
63      }
64
65      mTextureRegionHashMap = new HashMap<String, TextureRegion>();
66
67      float mRegionCoords[] = new float[] {
68         0.0f, 1.0f, // top left (V2)
69         0.0f, 0.0f, // bottom left (V1)
70         1.0f, 1.0f, // top right (V4)
71         1.0f, 0.0f // bottom right (V3)
72      };
73
74      gl.glGenTextures(1, mTexture, 0);
75      gl.glBindTexture(GL10.GL_TEXTURE_2D, mTexture[0]);
76      gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MIN_FILTER, GL10.
            GL_NEAREST);
77      gl.glTexParameterf(GL10.GL_TEXTURE_2D, GL10.GL_TEXTURE_MAG_FILTER, GL10.
            GL_LINEAR);
78
```

```
79      GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bitmap, 0);
80
81      bitmap.recycle();
82
83      mTextureRegionHashMap.put("default", new TextureRegion(mRegionCoords));
84
85      for (int i = 0; i < dataset.length(); i++)
86      {
87          int x = (int) ((i % mItemsInRow) * mCellWidth);
88          int y = (int) ((i / mItemsInRow) * mCellHeight + fontSize / 2);
89
90          this.addTextureRegion(dataset.substring(i, i + 1), new Point(x, y), new
                Size(mCharWidthMax, mCharHeight));
91      }
92  }
```

Kod źródłowy 4.24 Implementacja wczytywania czcionki.

Powyższy kod prezentuje sposób implementacji wczytywania tekstur. Konstruktor klasy jako argumenty przyjmuje referencję do obiektu biblioteki graficznej, nazwę, krój, wielkość czcionki oraz znaki, z których użytkownik będzie mógł korzystać. W początkowych liniach przypisywane są dane do zmiennych wewnątrz klasy oraz tworzone zmienne pomocnicze.

W liniach od 18 do 36 występuje pętla `for`, w której zapisywane są informacje dotyczące długości poszczególnych znaków znajdujących się w zmiennej `dataset`. W trakcie tego procesu, odnajdywana jest również maksymalna szerokość jaka występuje.

Znając maksymalną szerokość znaku utworzone zostają zmienne, które przechowują dane dotyczące wielkości elementu, do którego zapisywany będzie każdy znak. W tym momencie należy wyznaczyć rozmiar tekstury, która pomieści wszystkie elementy.

Należy pamiętać, iż biblioteka OpenGL określa, iż teksturą jest obraz, którego wymiary są potęgą liczby 2. Do określenia rozmiaru obliczamy pole jakie zajmować będą wszystkie znaki. Wartość tą pierwiastkujemy i zwiększamy tak, aby była ona wielokrotnością liczby dwa. W linii numer 45 sprawdzamy czy bitmapa o takich rozmiarach pomieści wszystkie elementy, jeśli nie to zwiększamy jej szerokość dwukrotnie.

W linii numer 53 tworzymy bitmapę o wyznaczonej szerokości oraz wysokości. Następnie zapisujemy do niej wszystkie znaki jakie znajdują się w zmiennej `dataset`. Po tej operacji analogicznie jak w klasie `Texture` tworzymy region oraz bindujemy teksturę.

Ostatecznie usuwamy pamięć bitmapy i dodajemy wszystkie znaki jako regiony do stworzonej tekstury. Dzięki temu jesteśmy w stanie odnaleźć niemal natychmiast dowolny znak, który został wczytany.

Przykłady użycia

- Wczytywanie czcionek do obiektu klasy Font.

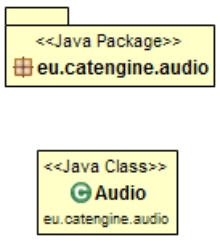
```
1  @Override
2  protected void loadResources()
3  {
4     getFontManager().loadFont("Verdana.ttf", 15);
5     getFontManager().loadFont("Verdana.ttf", 22);
6  }
7
8  public DemoScene(String sceneName)
9  {
10    // ...
11
12    Font verdana15pt = getFontManager().get("Verdana", 15);
13    Font verdana22pt = getFontManager().get("Verdana", 22);
14
15    Text text1 = new Text("Demo␣text␣using␣verdana␣15pt", new Point(100, 100),
          verdana15pt);
16    Text text2 = new Text("Demo␣text␣using␣verdana␣22pt", new Point(100, 800),
          verdana22pt);
17
18    getSceneTree().addObject(text1);
19    getSceneTree().addObject(text2);
20 }
```

Kod źródłowy 4.25 Przykład użycia klasy Font.

Powyższy fragment kodu prezentuje w jaki sposób należy wczytywać obiekty klasy `Font`. Metoda `loadResources()` pochodzi z klasy rozszerzającej `CatEngineCore` i wczytuje czcionki o zadanym rozmiarze do managera.

Konstruktor klasy `DemoScene` pobiera wczytane czcionki z managera. Następnie tworzy dwa obiekty typu `Text`, które wyświetlają na ekranie urządzenia tekst różniej wielkości z informacją jaka została użyta czcionka oraz jej wielkość.

4.1.8 Obsługa dźwięków



Rysunek 4.11 Diagram przedstawiający elementy pakietu eu.catengine.audio.

Pakiet `eu.catengine.audio` zawiera elementy odpowiedzialne za obsługę pojedynczych dźwięków w aplikacji z wykorzystaniem silnika gier.

4.1.8.1 Audio

Klasa `Audio` przechowuje pojedyncze dźwięki, którymi można zarządzać w trakcie działania aplikacji. Posiada metody do rozpoczęcia, zatrzymania oraz wstrzymania odtwarzania dźwięku. Dla każdego obiektu można ustalić inną wartość głośności.

Przykłady użycia

- Wczytywanie dźwięków do obiektu klasy Audio.

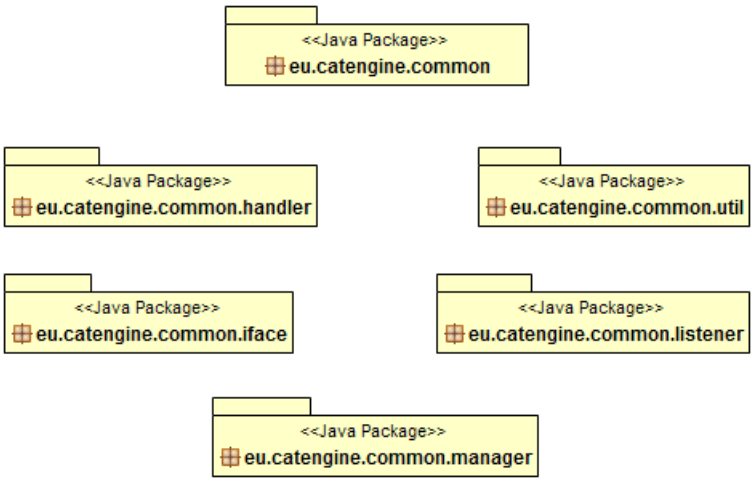
```
1  @Override
2  protected void loadResources()
3  {
4     getAudioManager().loadAudio("demo-audio.mp3");
5  }
6
7  @Override
8  public void onPointerDown(TouchPointer pointer)
9  {
10    getAudioManager().get("demo-audio").play();
11 }
```

Kod źródłowy 4.26 Przykład użycia klasy Audio.

Powyższy fragment kodu prezentuje w jaki sposób należy wczytywać obiekty klasy `Audio`. Metoda `loadResources()` pochodzi z klasy rozszerzającej `CatEngineCore` i wczytuje dźwięk znajdujący się w zadanej ścieżce do managera.

Metoda `onPointerDown(...)` znajdująca się w klasie `DemoScene` pobiera wczytany dźwięk oraz odtwarza go w momencie dotknięcia ekranu dotykowego.

4.1.9 Klasy ogólnodostępne



Rysunek 4.12 Diagram przedstawiający elementy pakietu eu.catengine.common.

Pakiet `eu.catengine.common` zawiera elementy dostępne dla wszystkich pozostałych klas silnika.

4.1.9.1 Handlery

Elementy zawarte pakiecie `eu.catengine.common.handler` odpowiadają za obsługę pewnych zdarzeń, które wywoływane są w trakcie pracy silnika. W obecnym momencie zawiera on tylko jedną klasę, która opisana jest poniżej.

4.1.9.1.1 TouchHandler

Klasa `TouchHandler` służy do obsługi zdarzeń, które wystąpiły w związku z dotknięciem ekranu urządzenia. Wywołuje następujące metody w aktywnej scenie:

- `onPointerDown(...)` - w momencie dotknięcia ekranu,
- `onPointerMove(...)` - w momencie poruszania wskaźnikiem po ekranie,
- `onPointerUp(...)` - w momencie odsunięcia wskaźnika od ekranu.

Ważne elementy

- Obsługa wielu punktów dotyku.

```
1  public void TouchAction(MotionEvent motionEvent)
2  {
3     mAction = motionEvent.getAction();
4     mActionCode = mAction & MotionEvent.ACTION_MASK;
5
6     mActiveScene = mSceneManager.getActiveScene();
7     mCameraPoint = mActiveScene.getCamera().getAnchorPoint();
8
9     for (int i = 0; i < motionEvent.getPointerCount() && i < mMaxPoints; i++)
10    {
11        mPid = motionEvent.getPointerId(i);
12
13        Point rawTouchPoint = new Point(motionEvent.getX(i), motionEvent.getY(i));
14        Point hudTouchPoint = mScreenManager.getHudPoint(rawTouchPoint);
15        Point sceneTouchPoint = mScreenManager.getScenePoint(mCameraPoint,
              rawTouchPoint);
16
17        mHistoricalTouchPointers[mPid] = mTouchPointers[mPid];
18        mTouchPointers[mPid] = new TouchPointer(mPid, rawTouchPoint, hudTouchPoint
              , sceneTouchPoint);
19    }
20
21    switch (mActionCode)
22    {
23       case MotionEvent.ACTION_DOWN:
24       case MotionEvent.ACTION_POINTER_DOWN:
25          mPid = mAction >> MotionEvent.ACTION_POINTER_ID_SHIFT;
26          mActiveScene.onPointerDown(mTouchPointers[mPid]);
27          break;
28       case MotionEvent.ACTION_MOVE:
29          for (int i = 0; i < motionEvent.getPointerCount(); i++)
30          {
31             mPid = motionEvent.getPointerId(i);
32             mActiveScene.onPointerMove(mTouchPointers[mPid],
                   mHistoricalTouchPointers[mPid]);
33          }
34          break;
35       case MotionEvent.ACTION_UP:
36       case MotionEvent.ACTION_POINTER_UP:
37       case MotionEvent.ACTION_CANCEL:
38       case MotionEvent.ACTION_OUTSIDE:
39          mPid = mAction >> MotionEvent.ACTION_POINTER_ID_SHIFT;
40          mActiveScene.onPointerUp(mTouchPointers[mPid]);
41    }
42 }
```

Kod źródłowy 4.27 Obsługa dotyku wielopunktowego w klasie TouchHandler.

Powyższy fragment kodu prezentuje w jaki sposób obsługiwany jest wielopunktowy dotyk na ekranie urządzenia. Metoda `TouchAction(...)` wywoływana jest automatycznie z klasy rozszerzającej `CatEngineCore`.

Na samym początku odczytany zostaje rodzaj wykonanej akcji oraz aktywna scena wraz z kamerą. Następnie w pętli `for` iterujemy po maksymalnie tylu elementach ile określa zmienna `mMaxPoints`, jednak nie więcej niż wartość zwrócona przez metodę `motionEvent.getPointerCount()`, która określa aktualną ilość wskaźników na ekranie.

Wewnątrz pętli `for` odczytywane jest id wskaźnika oraz tworzone są trzy punkty:

- `rawTouchPoint` - punkt wskaźnika bez żadnych zmian,
- `hudTouchPoint` - punkt wskaźnika dostosowany tak, aby określał pozycje elementów HUD,
- `sceneTouchPoint` - punkt wskaźnika dostosowany tak, aby określał pozycje elementów sceny.

Ostatni obiekt zapisany w tablicy `mTouchPointers` o indeksie `mPid` zostaje przypisany do tablicy `mHistoricalTouchPointers`. Po czym zostaje nadpisany przez nowy obiekt klasy `TouchPointer` z punktami opisanymi powyżej. Ostatecznie w zależności od zmiennej `mActionCode` wywoływane są odpowiednie metody w aktywnej scenie.

4.1.9.2 Przydatne narzędzia

Elementy zawarte w pakiecie `eu.catengine.common.util` służą jako dodatki dla programisty. Dzięki nim wykonywane są proste zadania, które najprościej zawrzeć w oddzielnych klasach.

4.1.9.2.1 Interpolator

Klasa `Interpolator` służy do zmiany pewnych parametrów obiektów sceny w zadanym czasie. Dane jakie są poddawane interpolacji to kolor, pozycja, obrót oraz skala obiektu. Klasa ta zawiera wewnętrzną klasę, którą można wykorzystać do rozbudowy działań o inne, bardziej skomplikowane efekty.

Przykłady użycia

- Zmiana danych obiektu sceny w zadanym czasie.

```
1  @Override
2  public void onPointerDown(TouchPointer pointer)
3  {
4     Random random = new Random();
5     Rectangle rectangle = (Rectangle) getSceneTree().getObject("demo-object");
6
7     if(rectangle.isAtPoint(pointer.getSceneTouchPoint()))
8     {
9         getInterpolator().rotateBy(rectangle, 45, 1000);
10        getInterpolator().colorTo(rectangle, new Color(random.nextFloat(), random.
              nextFloat(), random.nextFloat()), 1000);
11    }
12 }
```

Kod źródłowy 4.28 Przykład użycia klasy Interpolator.

Powyższy fragment kodu prezentuje przykładowy sposób w jaki należy wykorzystywać obiekt klasy `Interpolator` w dowolnej scenie. W momencie dotknięcia ekranu wskaźnikiem wywoływana jest metoda `onPointerDown(...)`. Wewnątrz niej sprawdzane jest czy użytkownik dotknął ekranu w miejscu, gdzie znajduje się obiekt `rectangle`. Jeśli tak, to pobieramy interpolator i wywołujemy zmianę rotacji oraz kolor dla obiektu.

4.1.9.2.2 FPSCounter

Klasa `FPSCounter` oblicza ilość wyświetlanych klatek na sekundę w ciągu ostatniej sekundy. Posiada metody statyczne, dzięki czemu nie trzeba tworzyć dodatkowego obiektu chcąc skorzystać z licznika.

Przykłady użycia

- Inicjalizacja licznika oraz wyświetlanie aktualnej wartości FPS.

```
1  public DemoScene(String sceneName)
2  {
3     // ...
4
5     FPSCounter.startCounter();
6
7     Text fps = new Text("Please␣wait...", new Point(0, 754), getFontManager().get
          ("Verdana", 15));
8     getHudTree().addObject(fps);
9  }
10
11 @Override
12 public void updateScene(float deltaTime, long elapsedTime)
13 {
14    FPSCounter.update();
15    fps.setText(FPSCounter.getFPS() + "");
16 }
```

Kod źródłowy 4.29 Przykład użycia klasy FPSCounter.

Powyższy kod prezentuje sposób w jaki należy wykorzystywać klasę `FPSCounter`. W konstruktorze sceny `DemoScene` należy zainicjalizować licznik poprzez użycie metody statycznej `FPSCounter.startCounter()`. Dodatkowo został utworzony obiekt klasy `Text`. Będzie on odpowiedzialny za wyświetlenie informacji pobranych z licznika. Metoda `updateScene(...)` aktualizuje stan licznika oraz zmienną `fps`.

4.1.9.3 Interfejsy

Elementy zawarte w pakiecie `eu.catengine.common.iface` to tylko i wyłącznie interfejsy występujące w silniku. Odnosząc się do języka Java, interfejs jest to zbiór deklaracji metod abstrakcyjnych oraz stałych i statycznych zmiennych. W odróżnieniu do deklaracji metod w klasie abstrakcyjnej, metody interfejsu muszą zostać uzupełnione o definicje w momencie ich użycia.

4.1.9.3.1 IColorable

Interfejs dla obiektów, które mogą zmieniać swój kolor. Posiada następujące metody:

- `setColor(...)` - przypisuje wartość koloru na podany jako parametr,
- `getColor()` - zwraca wartość koloru obiektu,
- `getRed()` - zwraca wartość czerwonej składowej koloru obiektu,
- `getGreen()` - zwraca wartość zielonej składowej koloru obiektu,
- `getBlue()` - zwraca wartość niebieskiej składowej koloru obiektu,
- `getAlpha()` - zwraca wartość składowej odpowiedzialnej za przezroczystość obiektu,
- `setColorListener(...)` - przypisuje obiekt nasłuchujący zmianę koloru.

4.1.9.3.2 IDisposable

Interfejs dla obiektów, które mogą zwolnić pewne zasoby w momencie ich usuwania. Posiada następujące metody:

- `dispose()` - metoda służąca do usunięcia pamięci po elementach znajdujących się wewnątrz obiektu.

4.1.9.3.3 IManager

Interfejs dla obiektów, które pełnią rolę managerów. Posiada następujące metody:

- `get(...)` - zwraca obiekt wczytany przez manager,
- `remove(...)` - usuwa obiekt z managera.

4.1.9.3.4 IMovable

Interfejs dla obiektów, które mogą zmieniać pozycję. Posiada następujące metody:

- `moveBy(...)` - zmienia położenie obiektów o zadany wektor,
- `moveTo(...)` - zmienia położenie obiektów do zadanego punktu,
- `setPositionListener(...)` - przypisuje obiekt nasłuchujący zmianę pozycji.

4.1.9.3.5 ISizable

Interfejs dla obiektów, które mogą zmieniać swój rozmiar. Posiada następujące metody:

- `getSize()` - zwraca wymiary obiektu,
- `getWidth()` - zwraca szerokość obiektu,
- `getHeight()` - zwraca wysokość obiektu,
- `setActualScaleAsNormal()` - przypisuje aktualną skalę jako początkową,
- `setScaleListener(...)` - przypisuje obiekt nasłuchujący zmianę skali,
- `scaleBy(...)` - skaluje obiekt o zadaną skalę względem punktu zaczepienia,
- `scaleByAtPoint(...)` - skaluje obiekt o zadaną skalę względem zadanego punktu,
- `scaleByAtCenter(...)` - skaluje obiekt o zadaną skalę względem jego środka,
- `scaleTo(...)` - skaluje obiekt do zadanej skali względem punktu zaczepienia,
- `scaleToAtPoint(...)` - skaluje obiekt do zadanej skali względem zadanego punktu,
- `scaleToAtCenter(...)` - skaluje obiekt do zadanej skali względem jego środka,
- `getScaleX()` - zwraca skalę obiektu dla osi X,
- `getScaleY()` - zwraca skalę obiektu dla osi Y.

4.1.9.3.6 IRotatable

Interfejs dla obiektów, które mogą zmieniać swoją rotację. Posiada następujące metody:

- `rotateBy(...)` - obraca obiekt o zadany kąt względem punktu zaczepienia,
- `rotateByAtPoint(...)` - obraca obiekt o zadany kąt względem dowolnego punkt,
- `rotateByAtCenter(...)` - obraca obiekt o zadany kąt względem jego środka,
- `setRotationListener(...)` - przypisuje obiekt nasłuchujący zmianę rotacji.

4.1.9.3.7 IRenderable

Interfejs dla obiektów niskiego poziomu, które mogą zostać wyświetlone w scenie. Posiada następujące metody:

- `setName(...)` - przypisuje nazwę obiektu na podaną jako parametr,
- `getName()` - zwraca nazwę obiektu,
- `setVisible(...)` - przypisuje widoczność obiektu w scenie,
- `isVisible(...)` - zwraca widoczność obiektu w scenie,
- `setVertices(...)` - przypisuje współrzędne wierzchołków obiektu,
- `getVertices()` - zwraca współrzędne wierzchołków obiektu,
- `setStartTime(...)` - przypisuje czas dodania obiektu do sceny,
- `getStartTime()` - zwraca czas dodania obiektu do sceny,
- `setUpdateNeeded(...)` - przypisuje informację, czy dany obiekt ma korzystać z wbudowanej metody `update(...)`,
- `isUpdateNeeded()` - zwraca informację, czy dany obiekt korzysta z wbudowanej metody `update(...)`,
- `setKeepInScene(...)` - przypisuje informację, czy dany obiekt ma nadal być przechowywany w scenie,

- `getKeepInScene(...)` - zwraca informację, czy dany obiekt ma nadal być przechowywany w scenie,
- `updateBuffer()` - aktualizuje bufor wierzchołków dla obiektu,
- `getAnchorPoint()` - zwraca punkt zaczepienia obiektu,
- `getCenterPoint()` - zwraca punkt środka obiektu,
- `isAtPoint(...)` - zwraca informację czy obiekt znajduję się w danym punkcie,
- `draw(...)` - renderuje obiekt,
- `update(...)` - aktualizuje obiekt, jeśli istnieje taka potrzeba.

4.1.9.3.8 IShape

Interfejs dla obiektów wysokiego poziomu, które mogą zostać wyświetlone w scenie. Implementuje metody zawarte w następujących interfejsach:

- `IRenderable`,
- `IMovable`,
- `ISizable`,
- `IColorable`,
- `IRotatable`.

4.1.9.3.9 ISprite

Interfejs dla obiektów graficznych, które mogą zostać wyświetlone w scenie. Posiada następujące metody:

- `setRegion(...)` - przypisuje region tekstury jako aktualny w użyciu,
- `getRegion()` - zwraca aktualny region tekstury obiektu,
- `setTexture(...)` - przypisuje teksturę obiektu,
- `getTexture()` - zwraca teksturę obiektu.

4.1.9.4 Listenery

Pakiet `eu.catengine.common.listener` zawiera tylko i wyłącznie klasy abstrakcyjne, które służą do nasłuchiwania pewnych zmian w danym obiekcie.

4.1.9.4.1 AnimationListener

Klasa nasłuchująca zmiany w działaniu animacji dla obiektów klasy `AnimtedSprite`. Posiada następujące metody:

- `onAnimationStarted(...)` - wywoływana w momencie uruchomienia animacji,
- `onNewAnimationCycle(...)` - wywoływana w momencie przejścia do nowego cyklu animacji,
- `onAnimationEnded(...)` - wywoływana w momencie zakończenia animacji.

4.1.9.4.2 AudioListener

Klasa nasłuchująca zmiany w odtwarzaniu dzięków dla obiektów klasy `Audio`. Posiada następujące metody:

- `onAudioStarted(...)` - wywoływana w momencie rozpoczęcia odtwarzania dźwięku,
- `onAudioPaused(...)` - wywoływana momencie wstrzymania odtwarzania dźwięku,
- `onAudioStopped(...)` - wywoływana momencie zakończenia odtwarzania dźwięku.

4.1.9.4.3 CameraListener

Klasa nasłuchująca zmiany pozycji dla obiektów klasy `Camera`. Posiada następujące metody:

- `onPositionChanged(...)` - wywoływana w momencie zmiany pozycji kamery,
- `onActionStopped(...)` - wywoływana w momencie ręcznego zatrzymania akcji,
- `onActionEnded(...)` - wywoływana w momencie zakończenia akcji.

4.1.9.4.4 ColorListener

Klasa nasłuchująca zmiany kolorów dla obiektów implementujących interfejs `IColorable`. Posiada następujące metody:

- `onColorChanged(...)` - wywoływana w momencie zmiany koloru obiektu.

4.1.9.4.5 PositionListener

Klasa nasłuchująca zmiany pozycji dla obiektów implementujących interfejs `IMovable`. Posiada następujące metody:

- `onPositionChanged(...)` - wywoływana w momencie zmiany pozycji obiektu.

4.1.9.4.6 RotationListener

Klasa nasłuchująca zmiany rotacji dla obiektów implementujących interfejs `IRotatable`. Posiada następujące metody:

- `onRotationChanged(...)` - wywoływana w momencie zmiany rotacji obiektu.

4.1.9.4.7 ScaleListener

Klasa nasłuchująca zmiany rozmiaru dla obiektów implementujących interfejs `ISizable`. Posiada następujące metody:

- `onScaleChanged(...)` - wywoływana w momencie zmiany skali obiektu.

4.1.9.4.8 InterpolatorListener

Klasa nasłuchująca zmiany w wykonywanych interpolacjach dla obiektów klasy `Scene`. Posiada następujące metody:

- `onInterpolationStopped(...)` - wywoływana w momencie ręcznego zatrzymania interpolacji,
- `onInterpolationEnded(...)` - wywoływana w momencie zakończenia interpolacji.

4.1.9.5 Managery

Pakiet `eu.catengine.common.manager` zawiera klasy managerskie, które służą do zarządzania danymi zewnętrznymi wczytanymi do pamięci. Managery jakie występują w silniku to:

- `AnimationManager` - manager odpowiedzialny za przechowywanie zasobów animacji,
- `AudioManager` - manager odpowiedzialny za przechowywanie zasobów dźwiękowych,
- `DataManager` - manager odpowiedzialny za przechowywanie dodatkowych danych aplikacji,
- `FontManager` - manager odpowiedzialny za przechowywanie zasobów czcionek,
- `SceneManager` - manager odpowiedzialny za przechowywanie scen,
- `ScreenManager` - manager odpowiedzialny za przechowywanie zasobów ekranu,
- `TextureManager` - manager odpowiedzialny za przechowywanie zasobów graficznych.

Każda z klas wymienionych powyżej korzysta z wzorca projektowego zwanego jako Singleton. Jego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji przy zapewnieniu globalnego dostępu do stworzonego obiektu.

Rozdział 5

Podsumowanie

Tworzenia silnika gier wiąże się z bardzo dużym nakładem pracy, a także nie da się jednoznacznie określić kiedy jest on skończony. Taki projekt można rozwijać przez wiele lat, dodając do niego kolejne funkcjonalności takie jak fizyka, systemy cząstek i tym podobne.

Moim zdaniem obecny silnik jest gotowy do tworzenia prostych oraz średniozaawansowanych gier, co za tym idzie jest skończony w stopniu, który chciałem osiągnąć. Gdybym posiadał więcej czasu z pewnością zaimplementowałbym system zderzeń lub podstawową fizykę sceny. Niestety w trakcie implementacji wyszło na jaw wiele problemów, które znacznie opóźniły finalizację. Część z nich zmusiła mnie do kilkukrotnej zmiany struktury projektu oraz nowego spojrzenia na sprawę. Jednym z największych problemów, z jakim się spotkałem było tworzenie drzewa sceny przechowującej obiekty. Musiałem stworzyć strukturę, która pozwala na szybki dostęp do danych, przy jednoczesnym zapewnieniu wygody użytkownika. Oprócz tego uboga dokumentacja na oficjalnej stronie platformy Android pokazywała jedynie podstawy wybranych zagadnień. W związku z tym bardzo dużo czasu musiałem poświęcić na szukanie informacji w innych źródłach, co nie zawsze się udawało. Idealnym przykładem braku jakiegokolwiek wzorca na oficjalnej stronie platformy jest obsługa dotyku wielopunktowego. Prac nie przyspieszało również tworzenie różnego rodzaju testów, które sprawdzały jakość zaimplementowanych metod oraz wykryły ewentualne błędy.

Jak wspominałem na wstępnie pracy tworzenie silnika gier wcale nie jest takie proste. Zwłaszcza na urządzenia mobilne. Pisząc tego typu silnik dysponuje się znacznie uboższą wersją biblioteki OpenGL. Co za tym idzie wszystkie elementy graficzne należało tworzyć prawie od podstaw. Obsługa grafiki została zaimplementowana dzięki bibliotece OpenGL 1.0 ES. W związku z tym aplikacje, które w nim powstaną będą obsługiwać urządzenia z systemem w wersji 2.2 lub wyższej. Stanowi to około 99% wszystkich urządzeń z systemem Android.

Rozdział 6

Załączniki do pracy

6.1 Spis tabel

3.1 Sprzedaż urządzeń mobilnych z system Android w latach 2012 oraz 2013. . . 12

3.2 Elementy generowane w czasie tworzenia projektu. 13

6.2 Spis rysunków

2.1 Uproszczony diagram działania biblioteki OpenGL. 7

2.2 Diagram obrazujący tworzenie trójkątów różnymi metodami. 9

2.3 Diagram obrazujący sposób rasteryzacji trójkąta o przypisanych kolorach wierzchołków. 9

3.1 Efekt wyświetlania obiektu klasy Rectangle. 18

3.2 Efekt wyświetlania obiektów klasy Rectangle oraz Sprite. 23

4.1 Diagram zależności między silnikiem i jego produktami. 25

4.2 Diagram przedstawiający elementy pakietu eu.catengine. 25

4.3 Diagram przedstawiający elementy pakietu eu.catengine.primitive. 29

4.4 Diagram przedstawiający elementy pakietu eu.catengine.renderable. 31

4.5 Diagram przedstawiający elementy pakietu eu.catengine.renderable.shape. . . 33

4.6 Diagram przedstawiający elementy pakietu eu.catengine.renderable.sprite. . . 35
4.7 Diagram przedstawiający elementy pakietu eu.catengine.renderable.text. . . 39
4.8 Diagram przedstawiający elementy pakietu eu.catengine.scene. 41
4.9 Scena implementująca grę *Pong*. 48
4.10 Diagram przedstawiający elementy pakietu eu.catengine.graphic. 51
4.11 Diagram przedstawiający elementy pakietu eu.catengine.audio. 58
4.12 Diagram przedstawiający elementy pakietu eu.catengine.common. 60

6.3 Spis kodów źrodłowych

3.1 Przykładowa aplikacja Hello World 13
3.2 Plik XML określający wygląd okna aplikacji 14
3.3 Plik XML z wartościami danych typu string. 14
3.4 Podstawowy kod obsługujący bibliotekę OpenGL. 14
3.5 Podstawowy kod obsługujący bibliotekę OpenGL cz.2. 16
3.6 Klasa Rectangle przygotowana dla środowiska OpenGL 17
3.7 Metoda zwracająca teksturę utworzoną z wczytanej bitmapy. 18
3.8 Sposób inicjalizacji zmiennych mGL oraz mAssetManager. 19
3.9 Podstawowa klasa Texture. 20
3.10 Podstawowa klasa Sprite. 21
4.1 Metoda określająca rozmiar okna aplikacji. 27
4.2 Przykład użycia klasy CatEngineCore. 27
4.3 Przykład użycia klasy Color. 29
4.4 Przykład użycia klasy Point. 30
4.5 Przykład użycia klasy Size. 30
4.6 Przykład użycia klasy Vector. 31
4.7 Przykład użycia klasy Renderable. 32

4.8 Przykład użycia klasy Line. 34

4.9 Przykład użycia klasy Rectangle. 34

4.10 Przykład użycia klasy Sprite. 36

4.11 Aktualizacja animacji w klasie AnimatedSprite. 36

4.12 Przykład użycia klasy AnimatedSprite. 38

4.13 Metoda rysowanie tekstu w klasie Text. 39

4.14 Przykład użycia klasy Text. 41

4.15 Implementacja pętli gry zawartej w Scenie. 42

4.16 Przykład użycia klasy Scene. 43

4.17 Przykład użycia klasy Layer oraz TreeLayer. 48

4.18 Przykład użycia klasy Camera. 49

4.19 Przykład użycia klasy TouchPointer. 50

4.20 Implementacja wczytywania tekstur. 51

4.21 Przykład użycia klasy Texture. 53

4.22 Przykład użycia klasy TextureRegion. 53

4.23 Przykład użycia klasy TextureAnimation. 54

4.24 Implementacja wczytywania czcionki. 55

4.25 Przykład użycia klasy Font. 58

4.26 Przykład użycia klasy Audio. 59

4.27 Obsługa dotyku wielopunktowego w klasie TouchHandler. 61

4.28 Przykład użycia klasy Interpolator. 62

4.29 Przykład użycia klasy FPSCounter. 63

6.4 Słownik pojęć

Activity - warstwa prezentacji programu. Zazwyczaj każde activity odpowiedzialne za osobne okno, które przy pomocy elementów View wchodzi w interakcję z użytkownikiem.
Anchor point - punkt, względem którego dokonywane są operacje dla danego obiektu.
Android - platforma oparta na jądrze Linux, w której skład wchodzi system operacyjny oraz SDK, które umożliwia tworzenie aplikacji z użyciem języka Java.
API - (ang. Application Programming Interface), interfejs programowania aplikacji, ściśle określony zestaw reguł oraz ich opisów wraz z gotowymi narzędziami.
Asset - zbiór zasobów zewnętrznych, które można wczytać do aplikacji.
Bitmap - obraz w grafice rastrowej, w którym zakodowany jest kolor każdego piksela.
FPS - (ang. Frames Per Second), ilość wyświetlanych klatek na sekundę.
GPU - (ang. Graphics Processing Unit), główna jednostka obliczeniową znajdującą się w nowoczesnych kartach graficznych.
Hash map - inaczej tablica mieszająca. Struktura danych, która jest jednym ze sposobów realizacji tablicy asocjacyjnej, w której zaimplementowano bardzo szybki dostęp do danych.
HUD - (ang. Head-Up Display), integralna część większości gier komputerowych zawierająca dodatkowe informacje na temat obiektu, lub obiektów sterowanych przez gracza.
Linux - uniksopodobny system operacyjny oparty o model wolnego oprogramowania.
OpenGL - (ang. Open Graphic Library), jest to wieloplatformowe API, które służy do renderowania grafiki 2D oraz 3D.
SDK - (ang. Software Development Kit), zestaw narzędzi dla programistów niezbędny w tworzeniu aplikacji.
Singleton - wzorzec projektowy, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji przy zapewnieniu globalnego dostępu do obiektu.
Sprite - dwuwymiarowy obiekt graficzny wyświetlany wewnątrz sceny.
URL - (ang. Uniform Resource Locator), ujednolicony format adresowania zasobów stosowany w Internecie.
XML - (ang. Extensible Markup Language), rozszerzalny język znaczników przeznaczony do reprezentowania dowolnych danych w sposób strukturalny.

Rozdział 7

Bibliografia

- Jason Gregory: *Game Engine Architecture*. Taylor and Francis Group, 2009. ISBN: 978-1-4398-6526-2.
- Ian F. Darwin: *Android. Receptury*. Helion, 2013. ISBN: 978-83-246-6269-2.
- *http://developer.android.com/*.